Table of Contents

[Linq fundamentals (course) 2](#_Toc9541803)

[Syntactic sugar 3](#_Toc9541804)

[LINQ Deferred execution 4](#_Toc9541805)

[Linq Operators 5](#_Toc9541806)

[סינון 5](#_Toc9541807)

[סידור 5](#_Toc9541808)

[קבוצות 6](#_Toc9541809)

[Linq to XML 9](#_Toc9541810)

[TPL Async – Ian 9](#_Toc9541811)

[Tasks 15](#_Toc9541812)

[Developer notebook 15](#_Toc9541813)

[Debugging 15](#_Toc9541814)

[Enable Just My Code 15](#_Toc9541815)

[debugview 16](#_Toc9541816)

[Customize debugger display values 16](#_Toc9541817)

[Pseudovariables 16](#_Toc9541818)

[DebugDiag 16](#_Toc9541819)

[Procdump 16](#_Toc9541820)

[Visual Studio Profiler 17](#_Toc9541821)

[Perfview 17](#_Toc9541822)

[mdbg 17](#_Toc9541823)

[C# 17](#_Toc9541824)

[null-coalescing operator 17](#_Toc9541825)

[don’t call virtual method from constructor 17](#_Toc9541826)

[06.The caller information attributes.wmv 18](#_Toc9541827)

[שינוי מקום הפלט של הקומפיילר 18](#_Toc9541828)

[הרכבת מחרוזת מאנומרציה 19](#_Toc9541829)

[anonymous class 19](#_Toc9541830)

[Check unicode charactres 19](#_Toc9541831)

[Friends in C# 20](#_Toc9541832)

[Environment Variables 20](#_Toc9541833)

[WPF 20](#_Toc9541834)

[Super simple data binding 20](#_Toc9541835)

[Hiding controls 21](#_Toc9541836)

[Getting Started with JSON in C\_Sharp Using Json NET 21](#_Toc9541837)

[Json definition 21](#_Toc9541838)

[המרת מחרוזת גייסון לאובייקט 22](#_Toc9541839)

[המרת אובייקט למחרוזת גייסון 24](#_Toc9541840)

[Read with JsonTextReader 24](#_Toc9541841)

[Write with JsonTextWriter 24](#_Toc9541842)

[טיפול בשגיאות המרה בתהליך דה-סרליזציה 25](#_Toc9541843)

[Settings and Attributes 26](#_Toc9541844)

[Create json classes using NewtonSoft 27](#_Toc9541845)

[המרת מחרוזות גייסון לקלאסים של גייסון 28](#_Toc9541846)

[SOLID Principles of Object Oriented Design 30](#_Toc9541847)

[The Single Responsibility Principle (SRP) 30](#_Toc9541848)

[The Open-Closed Principle 33](#_Toc9541849)

[The Dependency Inversion Principle (DIP) 36](#_Toc9541850)

[Constructor Injection 37](#_Toc9541851)

[Understanding Inversion of Control, Dependency Injection and Service Locator 39](#_Toc9541852)

[Using Service locator 40](#_Toc9541853)

[Using Dependency Injection 40](#_Toc9541854)

[Design Patterns Library 2015 41](#_Toc9541855)

[03 Bridge Pattern 41](#_Toc9541856)

[04 - The Builder Pattern 42](#_Toc9541857)

[28 - Strategy Pattern 45](#_Toc9541858)

[06 The Command Pattern 48](#_Toc9541859)

[07 Composite Pattern 49](#_Toc9541860)

[09 Event Aggregator 49](#_Toc9541861)

[10 Facade Pattern 50](#_Toc9541862)

[11- Factory Pattern 50](#_Toc9541863)

# Linq fundamentals (course)

למדנו כבר על דליגייט אנונימי. והרי דוגמא:

IEnumabel<string> filteredList = cities.Where( delegage(string s)

{return s.StartsWith(“L”);}

);

כאן מצהירים על דליגייט שמקבל מחרוזת ומחזיר בוליאן.

הקומפיילר מבצע איטרציה על רשימת הערים ומגיש למתודה שבדליגייט כניסה אחת בכל פעם. הדליגייט מחזיר ערך בוליאני הגם שאין כאן שום הצהרה מפורשת על כך.

דרך קצרה ותמציתית לכתיבת אותו דליגייט סמוי

IEnumabel<string> filteredList = cities.Where(

s => s.StartsWith(“L”)

);

דוגמאות להגדרת דליגייט

Func<byte, byte, bool> CheckWildEqual1 = (x, y) => x == y;

Func<byte, byte, bool> CheckWildEqual3 = (x, y) => { return x == y;};

Func<byte, bool> CheckWildEqual2 **=** x => x == 0;

כולם מחזירים ערך בוליאני משום שזה הארגומנט השלישי בהגדרת הגנריק Func.

**Action** הוא דליגייט לשימוש כללי אשר אינו מחזיר ערך

**Func** הוא דליגייט לשימוש כללי אשר מחזיר ערך ועל כן חייב להיות לו טיפוס אחד לפחות. Func<int> אינו מקבל ערך ומחזיר int. Func<string, int> מקבל מחרוזת ומחזיר int

כאשר הלינק הוא כנגד בסיס מידע, לא פונים לבסיס המידע שוב ושוב רק כדי לאמת ששורה מסוימת לא ממלאת תנאי, מהסיבה הפשוטה שיש בזה תקורה גבוהה מאד. התבנית Expression<> היא מבנה מיוחד. במקום לקמפל אותה לקוד IL הקומפיילר מפרק אותה ל"עץ ביטוי" (זה מה שנקרא code as data ומשמש ב remote linq), עץ ביטוי זה נשלח לבסיס המידע כך שהוא יודע לפענח את השאילתא ולבצע אותה בעצמו.

בעוד ש IEnumarable מקבל דליגייט רגיל, IQueryable מקבל Expression. זה הרבה פעמים שקוף למתכנת כי הקומפיילר עוטף את הביטוי שהמתכנת כתב בקלאס או בממשק הרצוי.

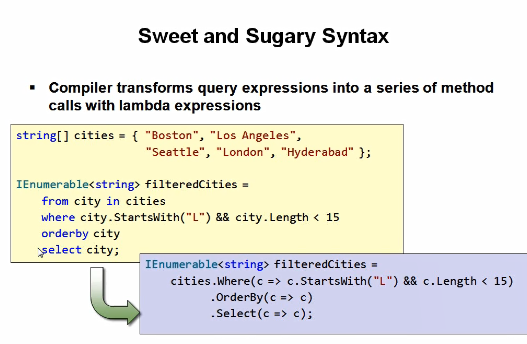
## Syntactic sugar

שתי צורות עקרוניות לכתיבת שאילתא של Linq

יש את comprehensive syntax זה בצהוב. (זה מה שנקרא syntactic sugar )

ויש את הכתיב הרגיל אליו רגילים המתכנתים. כתיב זה הוא עשיר יותר כי יש אופרטורים שאין להם מילת מפתח יעודית ב comprehensive syntax

בצילום הבא רואים את זה לעומת זה, שתי צורות הכתיבה שקולות והקומפיילר מתרגם את הראשונה לשנייה.



הצהרת ה Select אינה הכרחית בכתיב הרגיל היא כן הכרחית בכתיב המקיף. משתמשים בה כאשר רוצים הטלה Projection של הערך המוחזר.

## LINQ Deferred execution

הביטוי הבא אינו מתבצע בשלב הגדרתו

var list1 =

from item in rep.GetList()

where item.Name.StartsWith("G")

orderby item.Name

select item;

אלא כאשר משתמשים בו, כגון כאן:

foreach (var item in list1)

{

Console.WriteLine(item.Name);

}

קוראים לזה ביצוע דחוי ומילת המפתח שתומכת בזה היא yield

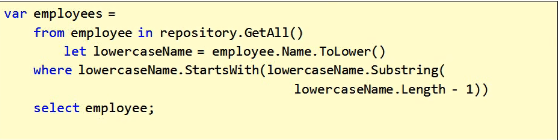
כלומר, מי שמימש את אופרטור Where השתמש ב yield - החזיר תוצאה אחת בכל פעם שפנו אליו ולא עבר על כל הרשימה.

אחת המשמעויות של ביצוע דחוי היא, שאם בין הגדרת הביטוי לבין השימוש בו ה repository ישתנה משום מה, שינוי זה ישתקף בתוצאה.

אופרטור שעובר על כל הרשימה כגון Count נקרא Greedy. הוספת ToList לביטוי הנ"ל הופכת אותו לגרידי.

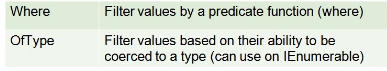
באופרטורים דחויים מבחינים בין כאלה שהם streaming לכאלה שהם לא. למשל orderby אמנם מחזיר פרט אחד בכל פעם כך שהוא דחוי אבל הוא חייב לעבור על כל הרשימה כדי שיוכל להחזיר את הפרט הנכון ואם הרשימה ארוכה מאד (או אינסופית, כלומר שהסוף שלה כלל לא מוגדר וכל מה שרוצים בכל פעם זה להביא את הפרט הבא בתור) אז אופרטור orderby לא יחזור זה non-streaming. לעומת זאת where חוזר ברגע שמצא פרט שעומד בקריטריון, הוא נחשב streaming

מילת המפתח let מאפשרת להגדיר range variable **נוסף**, אשר מיקל על ניסוח הביטוי



## Linq Operators

### סינון



אופרטור הסינון המקובל הינו **Where**, זהו Extension Method המוגדר על IEnumerable<T>

זו הגדרתו:

public static IEnumerable<TSource> Where<TSource>(

this IEnumerable<TSource> source, Func<TSource, bool> predicate);

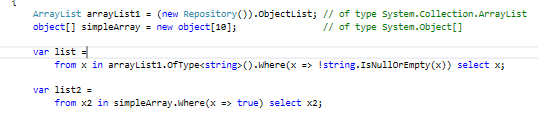
אבל לא תמיד האנומרציה מתבססת על IEnumerable<T>. לדוגמא, ArrayList מממש את IEnumerable, פירוש הדבר שלא ניתן להפעיל על אובייקט מטיפוס ArrayList את אופרטור Where.

יש אובייקט סינון נוסף, זהו **OfType** וזו הגדרתו:

Public static IEnumerable<TResult> OfType<TResult>(this IEnumerable source);

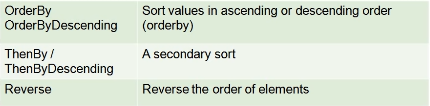
מקבל IEnumerable, מסנן לפי טיפוס, ומחזיר IEnumerable<T>

דוגמת קוד



OfType מחזיר IEnumarble<string> ועליו אפשר להפעיל את Where

### סידור



הכתיב המקיף שונה קצת בפרט כאשר מדובר בסידור משני

var v = Process.GetProcesses().

OrderBy(x => x.ProcessName).

ThenByDescending(x => x.MainWindowTitle).

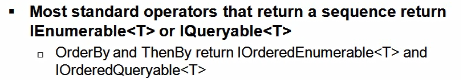
Select(x=>x.ProcessName)

var q = from v1 in Process.GetProcesses()

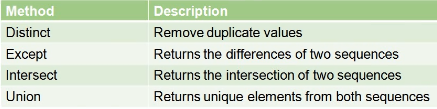
orderby v1.ProcessName, v1.MainWindowTitle

select v1.ProcessName;

הכתיב המקיף פשוט יותר. בכתיב הרגיל יש אבחנה ברורה בין OrderBy לבין ThenBy. אי אפשר להפוך בניהם משום ש OrderBy מחזיר IOrderedEnumerable<>



### קבוצות



#### Distinct

מסיר כפילויות מרשימה. פעולתו אינה תמיד טריוואלית. הוא משתמש במתודה Equals שבאה כברירת מחדל לכל קלאס, ומתודה זו מאמתת **שהאובייקטים** שווים. מה עם יש List<string> ורוצים להסיר כפילויות. מבחינת המתודה Equals כולם שונים למרות שהמחרוזת לכשעצמה יכולה להיות שווה.

אפשר להעמיס את מתודת Equals. זה לא מומלץ, א) כי צריך גם להעמיס את GetHash בהתאם וצריך למצוא פונקציה הש טובה וכו' ב) צריך להתחשב בעוד מקומות שמשתמשים במתודה זו.

דרך שניה, למסור לאופרטור Distinct קלאס השוואה אשר מממש את IEqualityComparer. זו דרך סבירה.

דרך שלישית היא לבצע הטלה של קלאס סתום בעזרת Select.

דוגמא

Process.GetProcesses().Select(x => new { x.PriorityClass, x.ProcessName }).Distinct();

נוצר כאן קלאס זמני רק לצורך ההשוואה ואופרטור ה Equals שלו מתייחס **רק למתודות הפומביות**.

#### Expect

מתייחס **לשתי** רשימות.

מחזיר את ההפרש. מבטיח שברשימה א' לא יהיה שום פרט המופיע ברשימה ב'

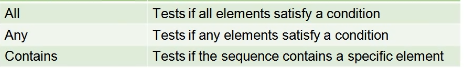
#### Intersect

מתייחס **לשתי** רשימות. פעולת חיתוך.

#### Union

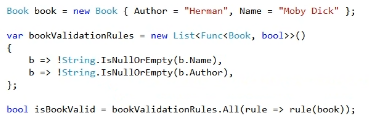
מתייחס **לשתי** רשימות. פעולת איחוד.

#### quantifiers



שים לב שבעוד ש ALL ו ANY מקבלים דליגייט, CONTAINS מקבל פשוט ערך.

בדוגמא הבאה, מגדירים רשימה של דליגטים, כל דליגט מגדיר תנאי, ובעזרת All מוודאים שאובייקט מסוים עומד ברשימת התנאים.



זה עשוי להיות יעיל מאד למשל כאשר באים לוודא שביטוי מסוים הוא חוקי ויש כמה היבטים לחוקיות זו.

#### Partitioning



דלג על 10 פריטים ראשונים והחזר את השאר. (לפעמים רוצים לקרוא קובץ csv שהשורה הראשונה שלו היא כותרת, רוצים לדלג על שורה זו **Skip(1)** מאד שימושי במקרה כזה).

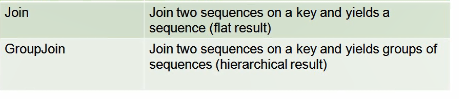
קח את 10 הראשונים והתעלם מהשאר.

דלג על אלה שעומדים בתנאי נתון

קח את אלה שעומדים בתנאי נתון.

מאד שימושי כאשר רוצים להציג רשומות, עמוד אחד המציע N רשומות בכל פעם.

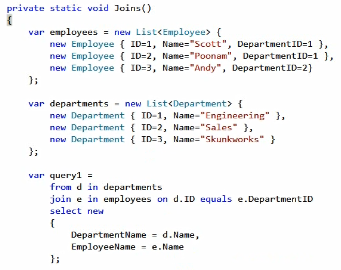
#### Join



איחוד של שתי רשימות לרשימה אחת על בסיס משותף.

למשל 'עובדים' ו'מחלקות'. איחוד שתי הרשימות לקבלת כל העובדים במחלקה **מסוימת**.

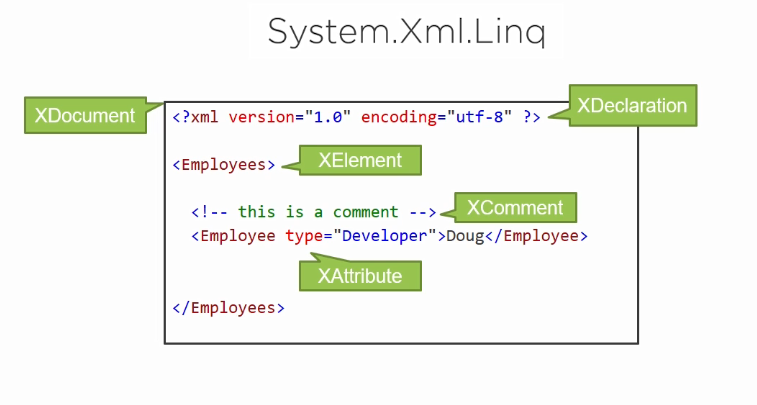
דוגמא



## Linq to XML

\LINQ Fundamentals with C# 6.0\07 LINQ to XML\02 System.Xml.Linq.mp4

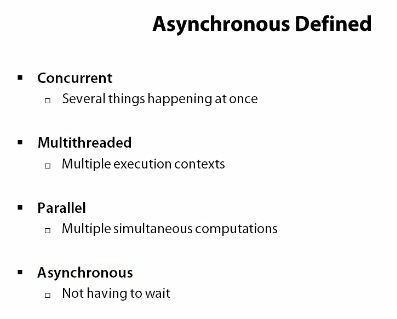
השילוב של לינק עם XML יצר מרחב שמות חדש. אם בעבר היה XmlDocument היום זה XDocument



כל הקלאסים הנ"ל יורשים מ XNode

# TPL Async – Ian

**pluralsight TPL Async**



Concurrent  
מושג כללי המתייחס לפעולות הנעשות במקביל. לא חשוב איך.

Multithread

מספר טרדים. יתכנו מספר טרדים ללא מקבילות משום שהם מבוצעים אחד אחרי השני.

Parallel

מתייחס למקבילות אמיתית.

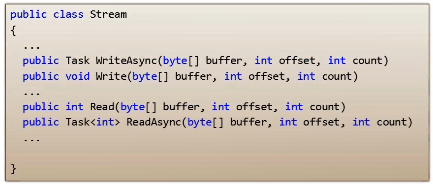
מובן מאליו שזה תלוי חומרה

Asynchronous

פעולה המתבצעת באופן אסינכרוני, וזה לאו דווקא כרוך בטרד. לדוגמא בפניה לדיסק **אין** טרד נוסף.

Task של דוט נט הוא שם כולל לפעולה אסינכרונית, שלאו דווקא כרוכה בפתיחת טרד.

קלאס Stream תומך ב Task אבל אין כאן טרדים.



משום שעצם הגישה לדיסק יוצרת השהייה.

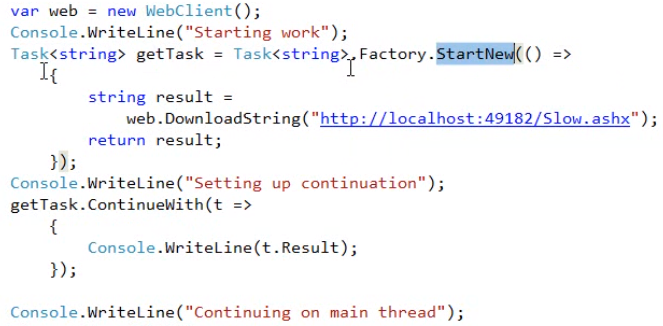
WriteAsync מחזיר Task. הוא ללא טיפוס מאחר והפעולה Write מחזירה כלום.

לעומת זאת ReadAsync מחזיר Task<int> (אשר יורש מ Task) מאחר והפעולה Read מחזירה int.

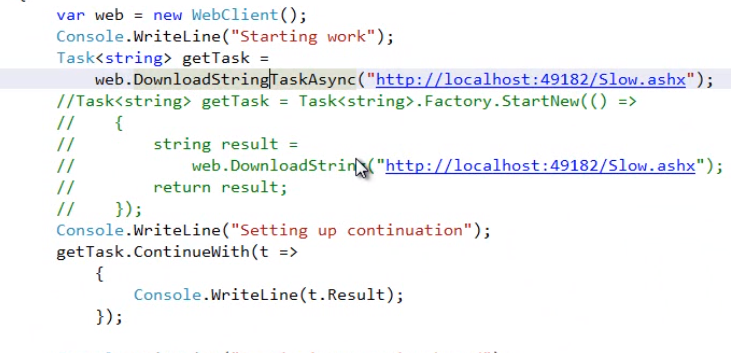
אותו int מוצב ב Task.Result (לסתם Task אין שדה Result). קריאת Task.Result זו פעולה חוסמת. היא משתחררת רק כשהטסק מסתיים.

קונספט נוסף ש TPL מציג זהו continuation. נאמר שיש פניה לדיסק שלוקחת זמן. בינתיים התוכנה ממשיכה בפעולה שגרתית ואגב כך בודקת האם הפניה לדיסק חזרה. זה יהיה מאד יעיל אם נוכל להצמיד טסק אל פעולת הפניה לדיסק כך שכאשר היא תסתיים הטסק יופעל אוטומטית תוך שהוא מקבל את תוצאת פעולת הפניה לדיסק ולא צריך לבדוק 'מדי פעם' האם הפעולה הסתיימה. זהו הרעיון של continuation.

זו דוגמא לפעולה שכרוכה ביצירת טרד והצמדת continuation.



וזו דוגמא לפעולה אסינכרונית **שלא כרוכה ביצירת טסק**, והצמדת continuation לטיפול בתוצאה. (מובן מאליו שעבור ה continuation המערכת כן מקצה טרד)



קורס זה מטפל ב tasks כללית ולאו דווקא כאלה שיש מאחוריהם טרד.

1. Asynchronous Programming with the TPL - Overview\07. Error Handling.wmv

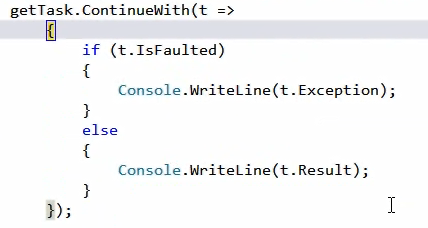
'משימה' עלולה להסתיים במצב חריג. גם זה נקרא סיום, אם יש continuation הוא יתבצע, אם מישהו מחכה עליו הוא ישתחרר אבל בזריקת חריג.

יש דרכים לבדוק את מצבו של Task

* IsFaulted
* Status
* Exception
* Result

האחרון הוא הוא האפשרות הטבעית ובמידה והטסק הסתיים עם חריג אז עצם הגישה ל Result תביא לזריקת חריג. אבל לא תמיד משימה מחזירה Result ומלבד זאת, לא בטוח שרוצים שהחריג יתפשט לקוד הקורא. לעיתים רק רוצים לדעת מהו. בעזרת IsFaulted אפשר לדעת האם הטסק הסתיים עם חריג ואז החריג מונח בשדה Exception

דוגמא לשימוש

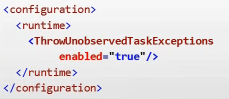


החריג של Task שקרס הוא לא סתם Exception אלא aggregateexception. יש כאן תמיכה לכמה חריגים ואף לשרשור של חריגים שהיא בהחלט יתכן שהמשימה הכילה תתי משימות ויש כאן כמה וכמה חריגים ורוצים לתפוס את כולם.

אם מתכנת אינו בודק את תוצאת ה task או את מצבו אלא פשוט נוטש אותו, ואותו task אכן זרק חריג לא מטופל. קוראים לזה **unobserved exception**. התוכנה לא תגיב לזה מיד אבל בשלב איסוף הזבל החריג יתעורר.

כברירת מחדל זה לא יגרום לאפליקציה לקרוס (לא רעיון טוב) והמשתמש כלל לא ידע שהייתה בעיה.

אפשר לשנות זאת:



הערה א: קשה לקבל קריסה כזו בזמן דיבאג משום שהקומפיילר דואג להשאיר התייחסות ל tasks והם לא נאספים. אם רוצים לבחון זאת צריך לקמפל במצב release ולהריץ ללא דיבאגר.

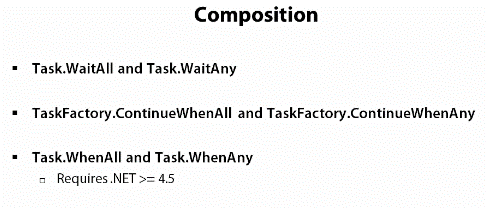
הערה ב: לא צריך ממש לקלוט את ה unobserved exception ב try-catch . מספיק שמתייחסים אליו כגון writeList(t.exception) מכאן והלאה אותו task לא יזרוק חריג גם כאשר הוא יאסף.

09. Continuations and Errors.wmv

אפשר להצמיד יותר מ continuation אחד ל task ואפשר להתנות מתי כל אחד ירוץ לדוגמא

OnlyOnFaulted, OnlyOnRanToComple and more….

10. Composition.wmv

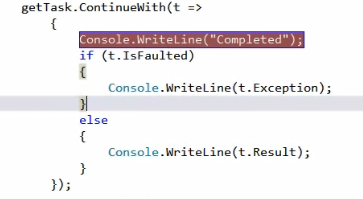


בעוד שהראשון והשני איכשהוא מובנים, השלישי צריך עיון

השלישי הוא עצמו task אשר ממשיך לכאורה את העבודה של ה tasks שקדמו לו



הוא מקבל את ה tasks הקודמים כארגומנטים והנה ההמשך:



הדגל isfaulted הוא במקרה זה or של השניים הקודמים.

השדה t.Exception הוא מטיפוס aggregateexception וכאן סיבה ברורה למה בכלל צריך aggregateexception

השדה t.Result הוא מערך של התוצאות של כל ה tasks הקודמים.

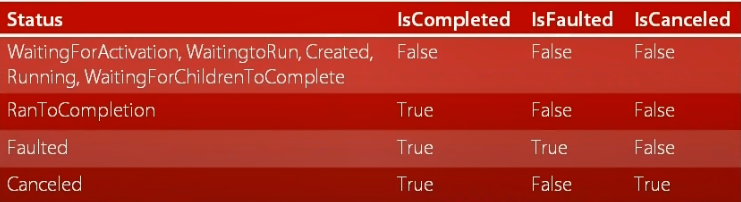
WhenAll מתאים יותר לטיפול אחיד בתוצאה של כמה משימות בעוד ש continueWhenAll מתאים יותר לטיפול אינדיבידואלי בכל משימה ומשימה.

12. Task Status.wmv

למשימה יש סטטוס. בדרך כלל לא משתמשים ישירות בשדה זה אלא ב

IsCompleted, IsFaulted, IsCanceled

שים לב לטבלה הבאה. משימה יכולה להיות 'מבוטלת' וגם 'מסוימת' בעת ובעונה אחת, אין סתירה בניהם. כך שכאשר שואלים שאלה, צריך לראות בדיוק את משמעותה.



02. TaskCompletionSourceT.wmv

כאשר משתמשים במתודה המחזירה Task כגון ReadAsync המתכנת לא יוצר בעצמו משימה. אם הוא כן רוצה ליצור משימה הוא צריך להשתמש ב Task.Run או ליצור אובייקט מטיפוס Task. בשני המקרים הוא צריך לספק דליגייט שיהיה בסיס ליצירת טרד, כלומר המתכנת יוצר טרד.

יש דרך שלישית. מה אם פונקציה Read לוקחת זמן אבל אינה מספקת ReadAsync. נכון שאפשר לייצר טרד אבל אפשר גם לחסוך יצירת טרד ועל כך בפרק זה.

03. Demo TaskCompletionSourceT.wmv



It seems like a recursive code but this is not.

FetchNextLine() is called and returns immediately.

Meanwhile, ReadLineAsync starts and continue with ProcessLine. Say on thread1.

(Async operation without a ‘Task’ continues with thread based task)

ProcessLine() starts, calls FetchNextLine() and returns immediately, thread1 **returns to pool**.

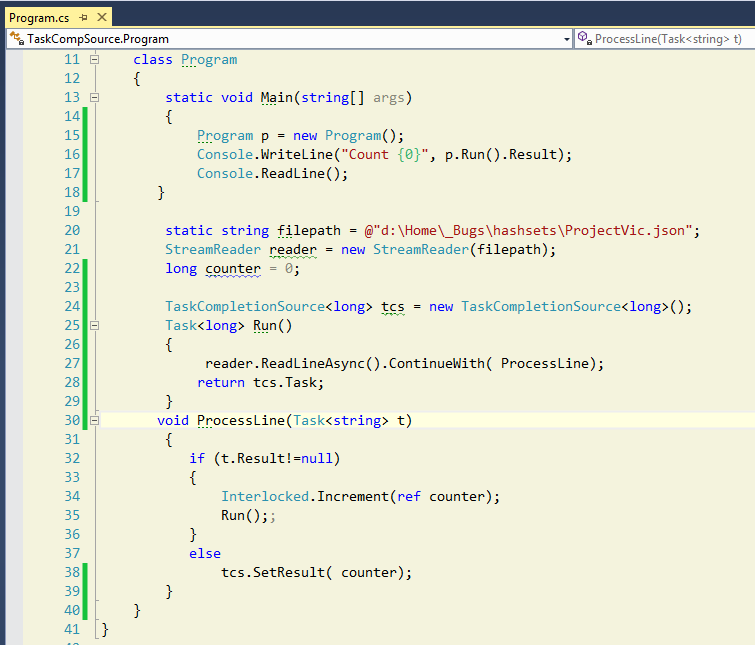
Meanwhile, FeatchNextLine cause ProcessLine to continue on thread2

ProcessLine returns immediately, thread2 returns to pool, and now thead3 will start …

We don’t have here a stack which grows! Thread by thread! And hence this it iterative, not recursive.

לפחות במקרה הספציפי הזה, לא נראה שיש רווח מכך שלא מקצים טרד יעודי לקריאת הקובץ. עדיף כל הזמן לקחת ולהחזיר טרד מהפול? ובוודאי שהקוד מורכב יותר עקב שימוש ב continuation וב TaskCompletionSource

הנה דוגמא שאני כתבתי לשימוש TaskCompletionSource



יש שלוש דרכים לקביעת תוצאה ל TaskCompletionSource

SetResult

SetException

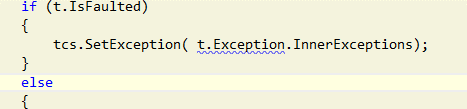
SetCanceled

pluralsight TPL Async - Ian #4.5\2. TaskCompletionSource\05. Demo Reporting Errors.wmv

מה קורה במקרה של חריג? ה TaskCompletionSource **לא** מכיר את החריג של ה task המקורי וכאשר הבקרה תחזור לקורא הוא לא ידע כלל שהמשימה לא הסתיימה כיאות.

יש להשתמש ב SetException וצריך לעשות זאת בצורה חכמה כי האובייקט שמועבר למתודה זו נעטף מיד ב aggregateexception ואם האובייקט עצמו הוא aggregateexception זה לא יראה טוב.

זו הדרך



המתודות הנ"ל, SetResult וכו' , ניתנות לקביעה פעם אחת בלבד. אי אפשר לקבוע פעמיים את תוצאת המשימה, ומי שינסה לעשות זאת יקבל מצב חריג. בתוכנות גדולות זה עלול לקרות ומשום כך יש מתודה משלימה

TrySetResult()

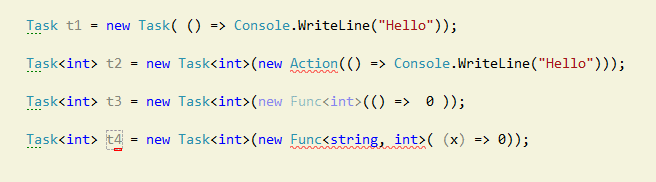
ובהתאמה לשתי המתודות הנוספות.

### Tasks

אתחול פשוט של משימה שאינה מחזירה כלום.



בדוגמא הבאה t1 מתחילים את המשימה אבל עדיין חסר Run כדי להתחיל ריצה



t2 כבר מחזיר מספר. ההצהרה של t2 **לא** חוקית => Action לא מחזיר ערכים.

ההצהרה של t4 גם לא חוקית. אמנם נעשה שימוש ב Func שמחזיר ערכים אבל הוא גם מקבל ארגומנט וזה לא תואם ל Task<int>

ההצהרה של t3 חוקית.

# Developer notebook

## Debugging

### Enable Just My Code

מסומן כברירת מחדל.

זה מונע דיבאג של קוד תשתיתי וכו'.

אז אם הדיבאגר לא נכנס או מסמן נקודה בעיגול חלול, יתכן שזו הבעיה.

זה מופיע גם כאשר מבצעים אנליזה של דמפ. השאלה שם האם להראות רק קלאסים שהמתכנת הגדיר או גם קלאסים מערכתיים.

### debugview

השקול ל

OutputDebugString ב C++

הינו

System.Diagnostics.Debug.WriteLine("I am using dot net debugging”);

System.Diagnostics.Trace.WriteLine("I am using dot net tracing”);

השני מוציא אל DebugView גם בגרסת Release

### Customize debugger display values

02.CUSTOMIZE DEBUGGER DISPLAY VALUES.WMV

כאשר מבצעים 'הובר' על אובייקט, רואים את ה ToString() שלו, שזה בדרך כלל שם מלא של הקלאס. לא כל כך שימושי. אם רוצים מידע על המופע פשוט להעמיס את ToString()

אם אי אפשר להעמיס את ToString(), אפשר להשתמש ב DebuggerDispaly. לדוגמא



באשר שדה Name הוא פשוט Prop של הקלאס.

DebuggerDispaly תקף לא רק לגבי קלאסים.

הערה: אפשר לשלוט איזה שדות של הקלאס יופיעו ואם יופיעו מהו המצב ההתחלתי. בעזרת DebuggerBrowsable

### Pseudovariables

אלה הם משתנים של meta-data. זה עשיר מאד ב C++ אבל גם ב C# יש את

$exception

$user

שאפשר לתת ב watch window

### DebugDiag

זהו כלי המאפשר לייצר דמפ כדי לנתח ביצועים או קריסה של אפליקציות. הוא מאד גמיש, מאפשר לקבוע את התנאים ליצור דמפ (רק כאשר צריכת הזיכרון עלתה מעל סף מסוים) ומאפשר לייצר קובץ דמפ כל פרק זמן מסוים.

הוא כולל גם כלי אנליזה חזק לדמפ, דברים שבדרך כלל מצויים רק בכלים שבתשלום.

חסרון: דורש התקנה ודומה שגם ההפעלה שלו (קביעת החוקים) קצת מסורבלת.

### Procdump

כלי ידוע מבית היוצר של sysinternals. בדומה ל debug-diag הוא גמיש בהגדרת התנאים. אינו דורש התקנה.

כלי זה מבוסס שורת פקודה וצריך לדעת מהם המתגים השונים.

בשורה התחתונה, אם אין סטודיו (בגרסאות המתקדמות 2015 ומעלה) כדאי להשתמש בכלי זה ליצור דמפ ובכלי האנליזה של dubug-diag כדי לנתח את התוצאות.

### Visual Studio Profiler

כאשר מבצעים אנליזה, צריך לבחור מראש את הספריות הנדרשות שבהם או מהן יתבצעו הדגימות

בחלון של Performance Explorer

בחר את תיקית Targets

קליק ימני

Add target project

ולבחור את ה dll העשויים להשתתף באנליזה.

### Perfview

קצת דומה ל Perfmon, אבל יותר יסודי ממנו. הוא אוסף 'מאורעות' בקצב גבוה יותר. התצוגה די בסיסית.

הוא לא בא עם מערכת הפעלה אבל הוא לא דורש התקנה

יש לסשה גולדשטיין מצגת יפה וגם למריו.

### mdbg

דיבגר לעניים. הוא ברמת קוד מקור אבל היכולת שלו להציג את שורות הקוד מוגבלת.

היתרון הוא שאין צורך בהתקנה כך שאם יש בעיה על מחשב לקוח ואין שם אפשרות להתקין סטודיו זה פתרון טוב.

הוא מסוגל לקבוע BP ברמת מתודות וברמת שורות, להראות את המשתנים בהקשר הנוכחי, לטייל מעלה מטה ב stack frame, וכל שאר הפעולות הנדרשות מדיבגר בסיסי.

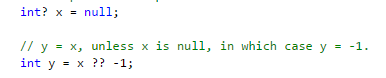
מאחר וכאמור יכולתו להראות את קוד המקור מוגבלת, צריך יהיה לראות את קוד המקור על מחשב אחר.

דיבגר זה כתוב ב C# וקוד המקור שלו ניתן להורדה ממיקרוסופט.

## C#

### null-coalescing operator

דרך נקיה לטפל בערכי null



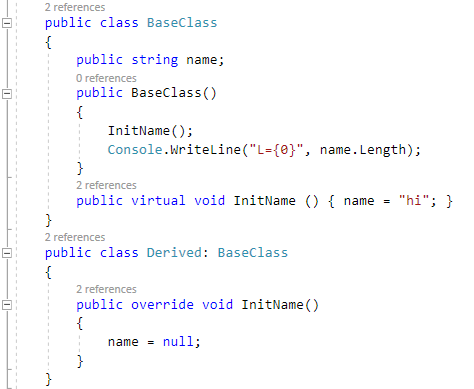
זה כמובן תקף לגבי אובייקטים.

### don’t call virtual method from constructor

05.THE DANGER OF VIRTUAL METHOD CALLS FROM CONSTRUCTORS.WMV

ב C++ קריאה למתודה וירטואלית מ ctor עלולה להיות הרסנית. אמנם ב C# המצב יותר טוב (ראה רשימה נפרדת) אבל גם ב C# עדיף שלא לעשות זאת.

בתוכנית זאת,



עצם האתחול של Derived יגרום לקריסה.

### 06.The caller information attributes.wmv

בעזרת [CallerMemberName], [CallerFilePath] and [CallerLineNumber]

יכולה מתודה לדעת מי קרא לה. תחליף מהיר ל StackFrame. שימושי במקרים מסוימים.

### שינוי מקום הפלט של הקומפיילר

את מקום קבצי היעד משנים דרך האופציות של הפרויקט עצמו. זה גורם לשינוי ב <OutputPath>

לדוגמא

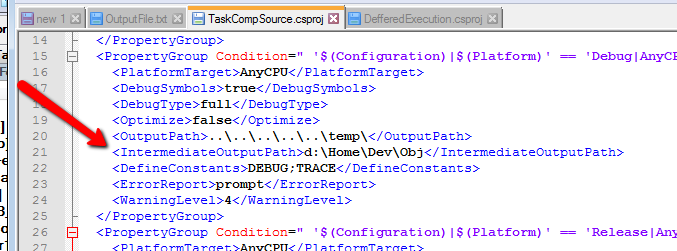
<OutputPath>..\..\..\..\Home\Dev\CS2017Sln\UsingTasks\_P2\bin\Debug\</OutputPath>

צריך לשנות גם ב debug וגם ב release .

כדאי לשמור על הקונבנציה של bin\debug

כדי לשנות את קבצי הביניים צריך לגעת ישירות בקובץ csproj.

בסמוך לתג <OutputPath> להוסיף תג <IntermediateOutputPath>

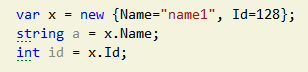


### הרכבת מחרוזת מאנומרציה

var s = string.Join("-", \_wildCardPattern.Select(x => x.Item1.ToString("X2")));

### anonymous class

למדנו כבר על דליגייט אנונימי ועל משתנה אנונימי (var). יש גם אפשרות לקלאס אנונימי



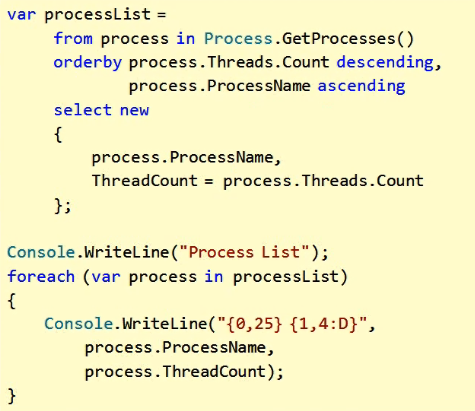
הקומפיילר יסיק בעצמו מהו הקלאס.

יש מספר הגבלות

* יורשים תמיד מ Object
* השדות שלהם הם פומביים
* הם תמיד Immutable , אי אפשר לשנותם לאחר הגדרתם.
* ניתן להגדירם רק בתחום המתודה אבל לא ברמת הקלאס. אי אפשר להחזירה ממתודה.

בפועל הם שימושיים בלינק כאשר משתמשים בכתיב מקיף comprehensive, לצורך ביצוע הטלה של התוצאה.

שים לב בתכנית הבאה להצהרה Select new



### Check unicode charactres

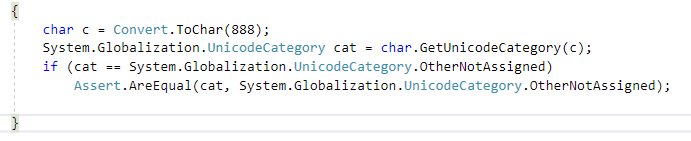
C# Tips and Traps\02.Part II\05.Testing char Unicode validity.wmv

ניתן לבדוק תווי יוניקוד ולראות לאיזה קטגוריות הם שייכים. לדוגמא:

* ClosePunctuation
* Control
* LineSeparator

הקטגוריה **OtherNotAssigned** בפרט אומרת שמדובר בתו יוניקוד לא תקין.

דוגמא לקוד שבודק תקינות של תו יוניקוד.



### Friends in C#

אמנם Friend הוא תכונה של C++ אבל מסתבר שגם ל C# יש מנגנון דומה.

[assembly:InternalsVisibleTo("Friend2a"),

InternalsVisibleTo("Friend2b")]

גודלו של אובייקט שהולך אל large object heap הוא 85000 בתים.

### Environment Variables

string folderPath = Environment.ExpandEnvironmentVariables(@"%systemroot%\Fonts\");

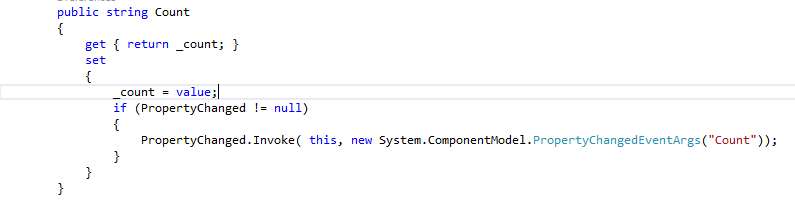
## WPF

### Super simple data binding

This should be declared in the code behind:



And now **Count** can be referenced from xaml:



The MainWindow must implement INotify



All left to do, is declare in xaml:



That’s it.

### Hiding controls

הסטייל הבא מחביא שדה במידה וערכו שווה ל false

(משמעות collapse בניגוד ל hidden שהוא לא תופס מקום ריק לאחר שקיפלו אותו)

במקרה הספציפי הזה מדובר על התצוגה ב Query Builder

<Style x:Key="UserMappingFieldStyle" TargetType="HeaderedContentControl" BasedOn="{StaticResource SimpleFieldStyle}">

<Style.Triggers>

<DataTrigger Binding="{Binding Fields[UserMapping].InstanceValue}" Value="False">

<Setter Property="Visibility" Value="Collapsed" />

</DataTrigger>

</Style.Triggers>

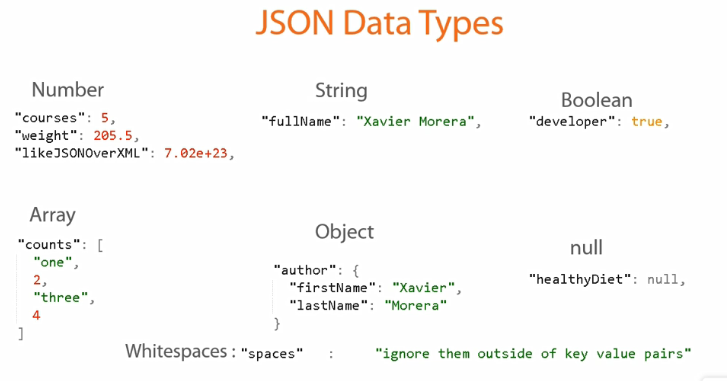
</Style>

# Getting Started with JSON in C\_Sharp Using Json NET

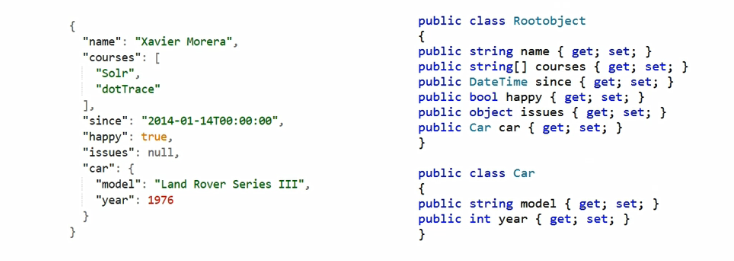
01\_07-JSON Specification.mp4

### Json definition

שלושה טיפוסי מידע, ושני טיפוסי הכלה:

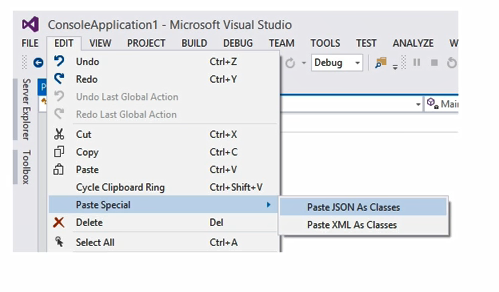


זו ההקבלה בין גייסון לבין קלאסים של C#



הסטודיו יודע להמיר טקסט גייסון לקלאס C#

מאד שימושי כאשר מקבלים קובץ\מחרוזת גייסון ורוצים לבנות קלאס מתאים.



### המרת מחרוזת גייסון לאובייקט

02\_03-Demo Serialize and Deserialize.mp4

הקלאס עצמו יוצר על ידי הסטודיו בטכניקה הנ"ל והשורה המסומנת יוצרת מופע של הקלאס מתוך מחרוזת גייסון.



### המרת אובייקט למחרוזת גייסון

String formattedJson = JsonConvert.SerializeObject(ro, Formatting.Indented);

הערה: הדגל indented דואג להכניס רווחים במקומות המתאימים כך שאם המחרוזת תוצג בקובץ, ניתן יהיה לקרוא אותה בקלות.

02\_04-Demo Object References.mp4

הסריאלייזר עובד לפי ערך. כלומר, אם נותנים לו מערך אובייקטים ובמערך זה יש הצבעה לאותו אובייקט משתי כניסות נפרדות, הוא יתייחס אליהם כשני אובייקטים נפרדים. ובשלב דה-סרליזציה יווצרו שני אובייקטים. יש דרכים להתגבר על זה. ראה במצגת.

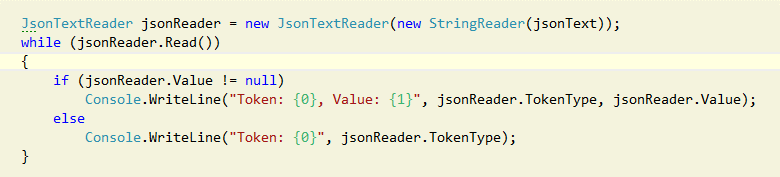
### Read with JsonTextReader

עבודה עם סרילייזר כרוכה בתקורה מסוימת. לעיתים קל יותר לעבוד עם

JsonTextReader

קלאס זה מקבל מחרוזת גייסון ומפרסר אלמנט אחר אלמנט. הוא רץ קדימה בלבד.

דוגמא:



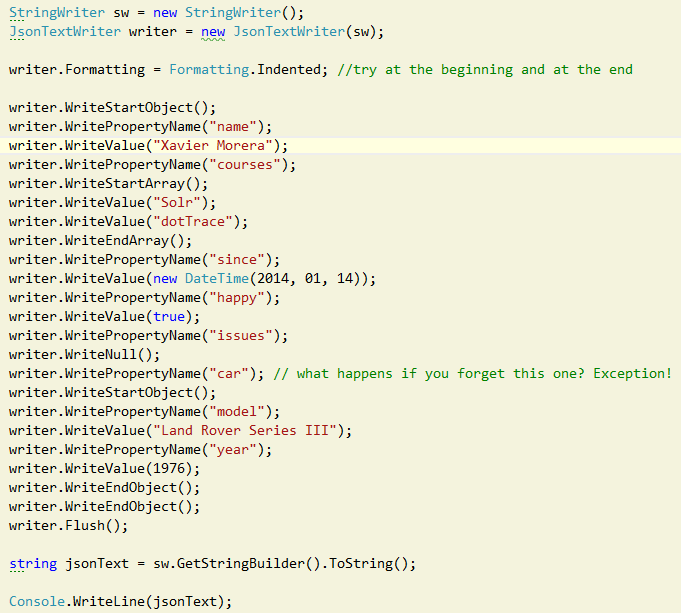
(לא כל כך שימושי)

### Write with JsonTextWriter

גם ביצירת מחרוזת גייסון יש דרך הכרוכה בפחות תקורה ועשויה להיות יעילה למחרוזות קטנות. הקלאס המקביל הוא

JsonTextWriter

דוגמא:



(גם זה לא כל כך שימושי)

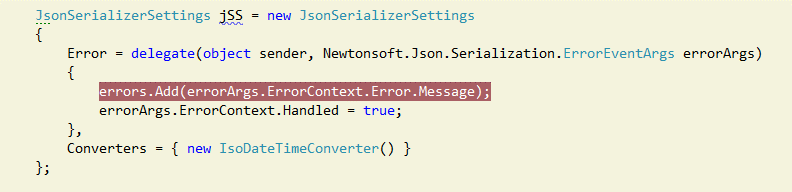
### טיפול בשגיאות המרה בתהליך דה-סרליזציה

02\_14-Dem Error Handling.mp4

מה קורה כאשר יש בעיות בטקסט שעובר דה-סרליזציה. נאמר שיש מערך המכיל תאריכים ואחד מהתאריכים בפורמט שגוי. האם בגלל זה כל התהליך יתעופף?

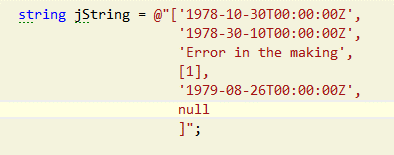
יש מוכנות למצבים מהסוג הזה.

הקטע הבא מאפשר להגדיר דליגייט אשר מטפל במצבי שגיאה ומאפשר ולאחר טיפול זה הדה-סרליזציה עשויה להמשיך.



שים לב ששדה Handled חייב לקבל true.

בהנחה שהטקסט אותו רוצים נראה כך

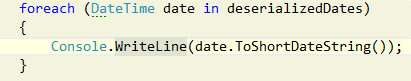


השורה הבאה תעשה את העבודה

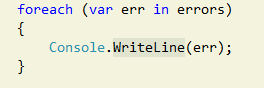


לאחר שורה זאת, רצוי לטפל גם במערך המכיל את הטעויות

זה הטיפול בתוצאות המוצלחות



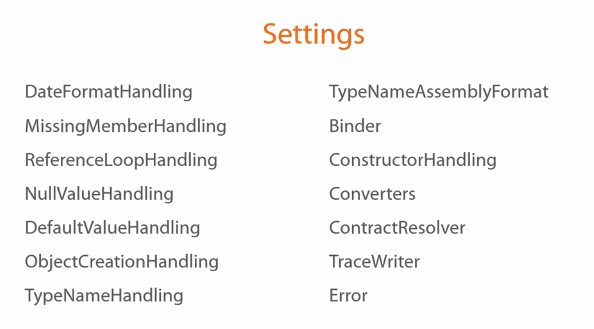
וזה הטיפול בשגיאות



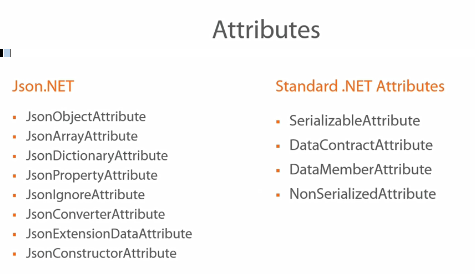
### Settings and Attributes

ספריית גייסון די גמישה ואפשר לתת הגדרות בהרבה מקומות, נתקלנו כבר ב Error וב Converters

הנה רשימה מלאה יותר



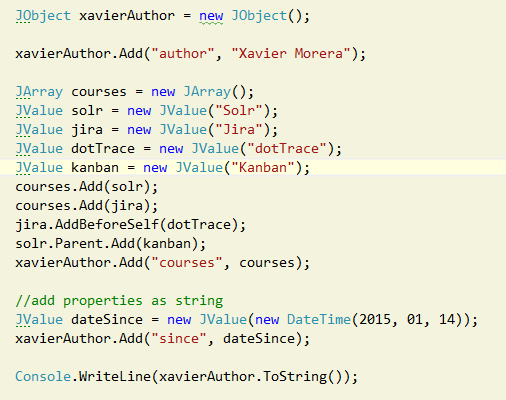
יש גם סדרה של אטריבוטים המשפיעים בעיקר על תהליך הסרליזציה.



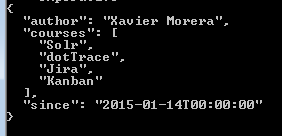
06\_03-Demo Create JSON Imperatively.mp4

### Create json classes using NewtonSoft

זו הדרך הפשוטה ליצירת גייסון באמצעות אובייקטים של NewtonSoft



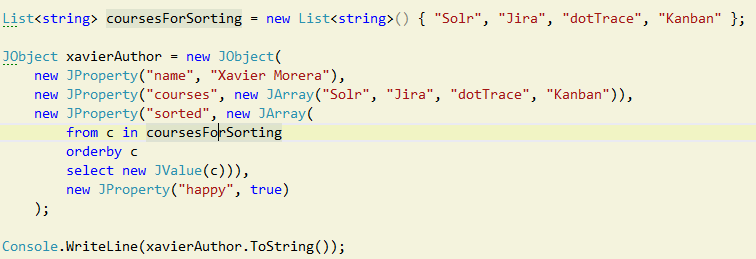
התוצר של פעולה זו הינו



האובייקט הראשי מגדיר את הסקופ של המחרוזת ויש שלש אלמנטים תחת האובייקט הראשי כשאחד מהם הוא מערך.

06\_04-Demo Create JSON Declaratively.mp4

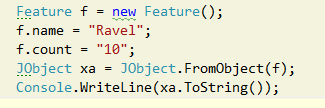
וזו דרך נוספת



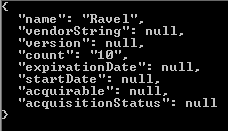
אולם הדרך הפשוטה ביותר הינה באמצעות FromObject

פשוט הופכים 'סתם' אובייקט לאובייקט גייסון.

זה הקוד



וזה התוצר



### המרת מחרוזות גייסון לקלאסים של גייסון

06\_07-Demo Parse JSON from String.mp4

למדנו כיצד להמיר מחרוזת גייסון לאובייקט של קלאס הידוע מראש באמצעות deserilzeObject. בהמרה זו לא נכנסים כלל לאובייקטים של גייסון כגון JObject.

כאן מדובר על המרת מחרוזות גייסון כאשר הקלאס אינו ידוע מראש.

ניתן לפרסר מחרוזת גייסון שלמה (ברמת האובייקט) או חלקית (מערך או אפילו טוקן בודד)

#### רמת הטוקן



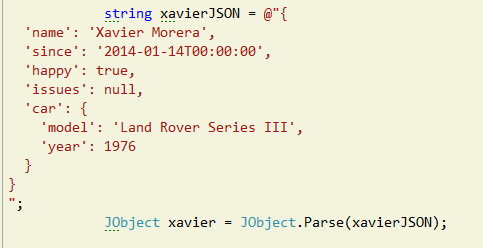
הערה JToken עצמו הוא אבסטרקטי, והמופע הינו מטיפוס JObject.

רמת המערך:

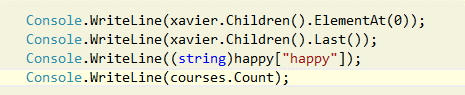


(הפסיק שיש בסוף לא מפריע, זה מערך עם ארבעה איברים).

והשימוש העיקרי הוא ברמת האובייקט



יש הרבה דרכים לחקור את JObject. הרי מקצת מהם:



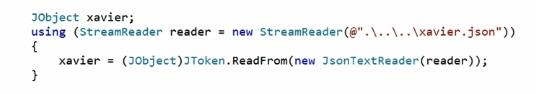
וכן



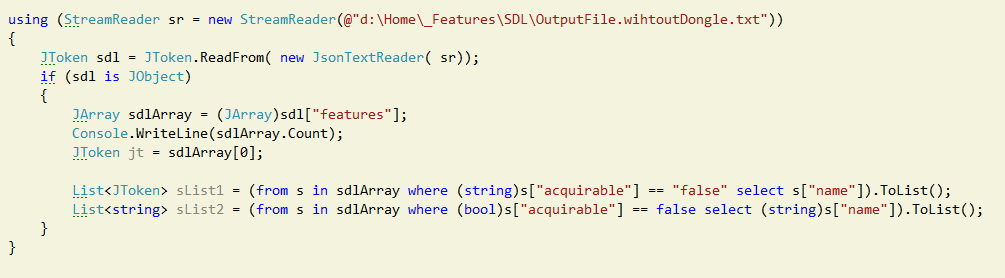
יכול לשמש לקבלת האובייקט המקונן, וכדי להגיע לאברי מערך משתמשים פשוט באינדקס courses[0]

אלה שיטות לחקירה של מבנים פשוטים ובהמשך יפורט יותר לגבי מבנים מורכבים.

קריאה ישירה מקובץ



06\_12-Demo Querying JSON with LINQ.mp4



לגייסון יש כמובן תמיכה בלינק. הדוגמא לעיל מראה שתי חקירות דומות מאד. S מייצג Jtoken ואפשר לבצע לו cast מאד גמיש.

בשורה הראשונה לוקחים אותו כמחרוזת ובשורה השניה כבוליאן. שניהם **אותו תנאי** .

גם ביצירת ההטלה, בשורה הראשונה לוקחים אותו כ JToken (למרות שהוא בעצמו JObject , תמיד מתייחסים לאב האבסטקרטי) ובשורה השניה לוקחים אותו כמחרוזת.

# SOLID Principles of Object Oriented Design

PS - Smith - SOLID Principles of Object Oriented Design #45

### The Single Responsibility Principle (SRP)



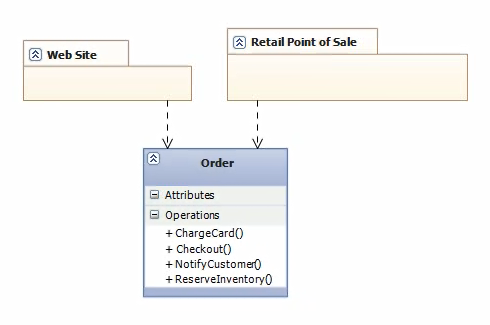
הגבלת הקלאס לתחום אחריות אחד

הפעילות הקשורה לאחריות זו צריכה להתבצע בקלאס עצמו ולא על ידי ספריות חיצוניות

תחום אחריות צר ככל האפשר מפחית את הסיבות לשינוי בקלאס. צריך שתהיה סיבה אחת בלבד לשינוי בקלאס [אמנם קשה לומר מהי סיבה 'אחת']

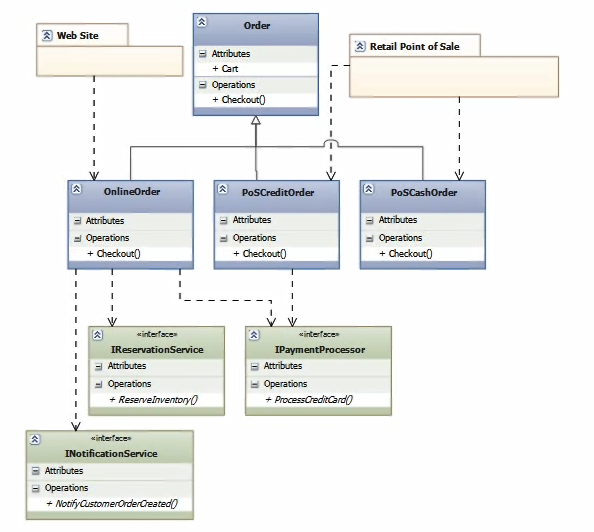
בדרך כלל דרישות מתמפות לתחומי אחריות

יש הקבלה: תחום אחריות = סיבה לשינוי



קלאס Order אחראי על 'ביצוע הזמנה'. אבל אחריות זו היא גדולה מדי לקלאס אחד.

שינוי במנגנון ההודעה ללקוח יגרום לקלאס להתקמפל מחדש ויגרום לשני הקלאסים התלוים בו להתקמפל מחדש אבל הקלאס שמממש את Point Of Sale לא מתעניין כלל בהודעות ללקוח שהרי הלקוח נמצא מול המוכר ונמצא שיש כאן תלות מיותרת.



הפתרון:

הפרדה לשלושה קלאסים: מכירה דרך הרשת, מכירה פיזית באשראי ומכירה פיזית במזומן.

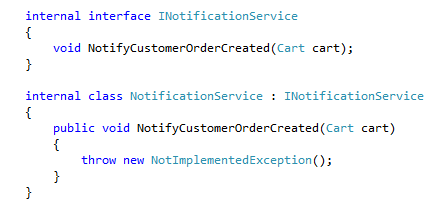
ויש כאן אלמנט נוסף. הפונקציונליות המשותפת לא רק שהופרדה לקלאסים , אלא גם נעשית דרך ממשקים. לדוגמא, ההודעה ללקוח נעשתה באמצעות המתודה

NotifyCustomer(cart)

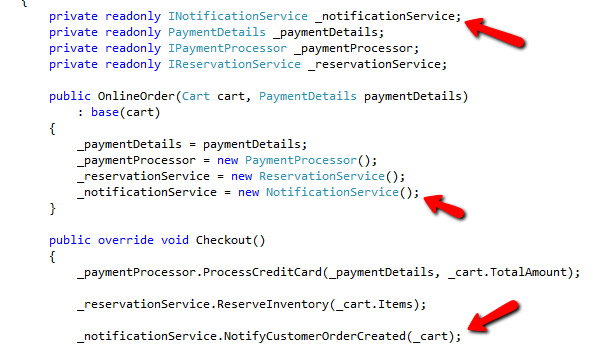
ואילו עכשיו המנגנון יותר מורכב אבל פחות נתון לשינויים

מגדירים ממשק וקלאס שמממש אותו. הקלאס המממש **הוא** זה שקורא ל NotifyCustomer(cart)

(למרות שכאן בציור אין לו מימוש)



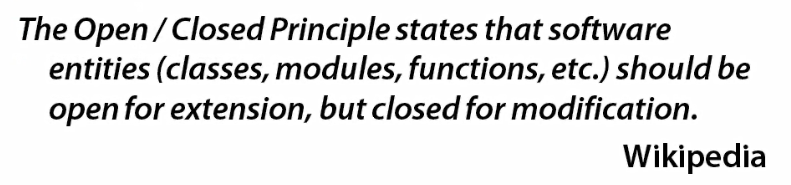
מי שרוצה להודיע ללקוח תלוי בממשק הנ"ל, וברמת ה ctor בלבד הוא תלוי גם במימוש ספציפי.



אם כן הרווחנו כאן שמכירה דרך הרשת ומכירה דרך נקודה פיזית תלויות רק באותם דברים הנוגעים אליהם, וגם המימוש עצמו של הקלאסים הוא בתלות רופפת באופן המימוש של הפעולות השונות.

### The Open-Closed Principle

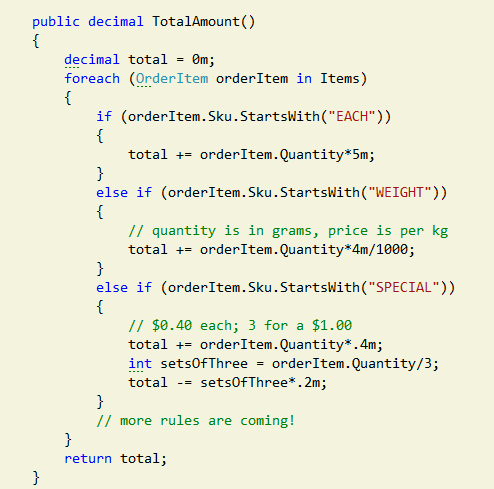
"פתוח סגור". פירוש, פתוח להוספה סגור לשינויים.



כאשר מוסיפים קלאסים לא צריך לבדוק מי משתמש בהם ומה השפעתם בכל מיני מקומות שהרי הם חדשים. לא צריך גם לשנות בדיקות יחידה.



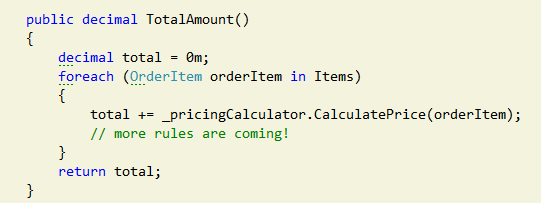
דוגמא



מתודה זו מחשבת סכום רכישה. החישוב הוא בהתאם לסוג הפריט מחיר וכמות. לכל סוג פריט sku צורת חישוב משלו.

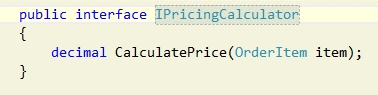
החיסרון הברור כאן הוא חוסר הגמישות לשינויים. שינוי בצורת חישוב אחת משפיעה על המתודה כולה וכל שכן הוספת צורת חישוב לסוג פריט חדש. שינוי מתודה עובדת ובדוקה זה דבר לא רצוי.

קודם כל, קוד בהיר יותר:



מכאן והלאה יש שימוש ב Strategy design pattern כלומר תלות בהפשטות (=ממשקים) במקום אובייקטים ממשיים.

נגדיר ממשק

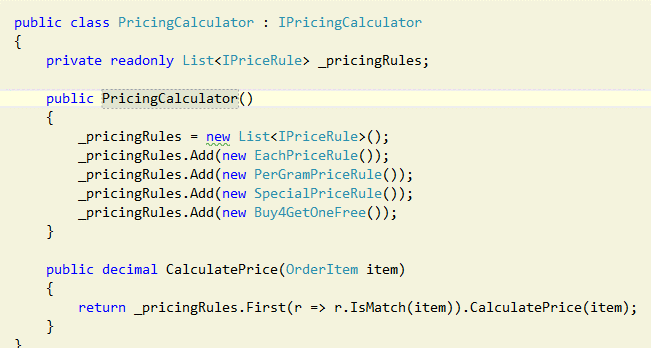


ונשתמש בו



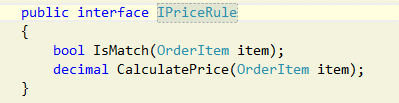
כך שהמתודה המקורית TotalAmount תהיה תלויה בממשק ולא במימוש.

המימוש עצמו של CalculatePrice , לאחר אתחול מתאים, נראה כך

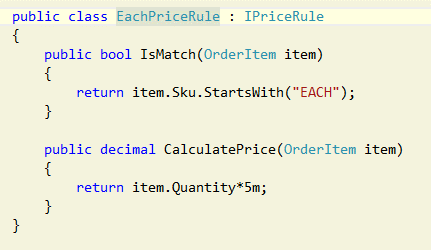


האוסף PricingRules מבוסס אל IPriceRule כך ששוב יש כאן תלות בהפשטה ולא במימוש.

IPriceRule מציג שתי מתודות. האחת, האם סוג הפריט מתאים לכלל זה, והשנייה, החישוב עצמו.



וזו דוגמא למימוש של PricingRule



(כאן כבר מונח החישוב שהיה במתודה המקורית)

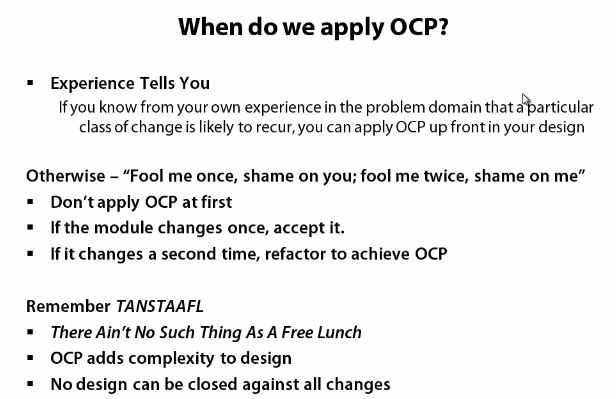
מה הרווחנו בזה?

שינוי כלל נעשה בקלאס אחד בלבד האחראי על כלל זה בלבד.

הוספת כלל כרוכה בהוספת קלאס ללא צורך בשינוים למעט ב ה PricingCalculator שבו צריך להוסיף שורה עבור הכלל נוסף. [ואם היינו משתמשים בטעינה דינמית גם ה PricingCalculator לא היה משתנה].

לסיכום,

לא תמיד צריך לרוץ ל OCP, משום שזה מוסיף מורכבות, אבל כשזה נדרש זה יעיל ביותר.



### The Dependency Inversion Principle (DIP)

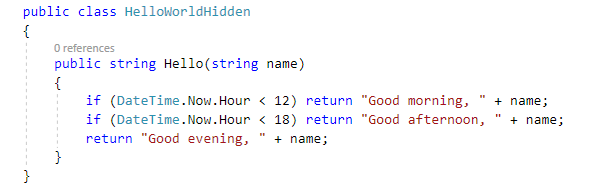
לקלאסים יש תלויות, בקלאסים אחרים, בספריות חיצוניות, בבסיסי מידע, בסוקטים וכיו"ב. תלויות אלה מכבידות ומסבכות כפי שנראה בהמשך, והגם שאין דרך להימנע מכך שקלאס ישתמש בתלויות כדי לבצע את מלאכתו, אפשר לשנות את צורת התלות באופן שמכביד פחות.

בעיות שאיתן מתמודדת תצורת DIP:

* כל שינוי במימוש של ה DB יגרור אחריו שינוי בקלאס. זה מה שנקרא tight coupling.
* חוסר אפשרות לשנות פרטים של אימפלמנטציה מבלי לגעת בגוף הקלאס. OCP.
* בדיקה או שימוש בהיבט מסוים של הקלאס שלא צריך בסיס מידע או שבסיס המידע לא משחק תפקיד. (האם הקלאס בכלל יסכים לרוץ על מחשב שאין בו בסיס מידע?)

קודם כל נגדיר שקלאס שמקבל ב ctor שלו את כל הספריות בהן הוא תלוי נחשב שיש לו תלויות מפורשות, וזה מצב הרבה יותר טוב מקלאס שתלוי תלויות סמויות.

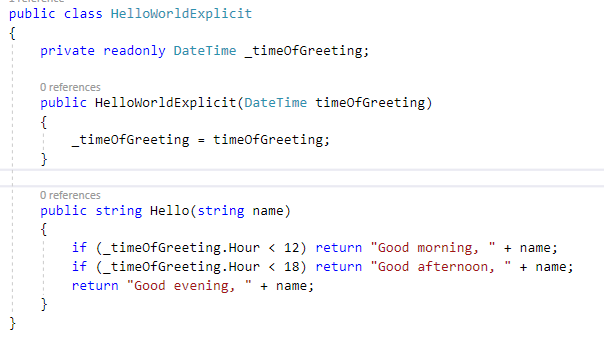
לדוגמא



קלאס זה תלוי באופן סמוי ב DateTime.

(יש כאן גם הפרה של OCP , ראה שם)

לעומת זאת כאן התלות מפורשת:



כאשר קלאס מקבל את התלויות ממי שקורא לו, אנו קוראים לזה **Dependency Injection**,

יש טכניקות למימוש Dependency Injection.

* **Constructor Injection** דחיפת המשאבים דרך ctor.
* Property (setter) injection
* Parameter injection
* **Service locator** (לא נידון בקורס זה)

### Constructor Injection

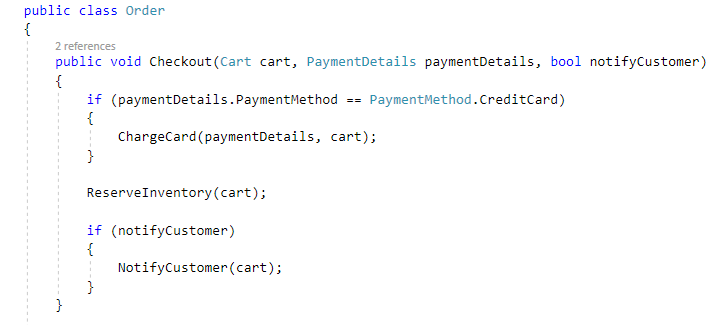
יתרונות

* ברור היטב במה הקלאס תלוי.
* ניתן להשתמש ב ioc container אבל אפשר גם להסתדר בלעדיו
* הקלאס נמצא במצב חוקי עם איתחולו

חסרונות

* אם יש הרבה תלויות, צריך הרבה ארגומנטים
  + כללית, הרבה ארגומנטים הם אינדיקציה לתכנון לא מתאים.
* יש מקומות שנדרש default ctor ואתחול בעזרתו שם את האובייקט במצב לא חוקי.
* יש מתודות שלא צריכות תלויות מסוימות
  + גם זה אינדיקציה למשהו לא טוב. פעולות הקלאס צריכות להיות 'הדוקות'. Cohesion.

שים לב לקלאס הבא אשר המתודה העיקרית שלו היא checkout.

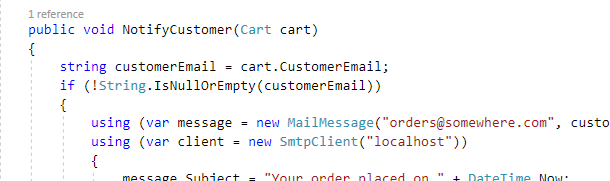


לקלאס זה יש תלויות סמויות. חיוב הכרטיס, שמירת פריט כנגד ההזמנה, הודעה ללקוח, כל אלה תלויים במנגנונים יעודיים.

לתהליך הrefactoring שיתואר כאן יש שלש מטרות

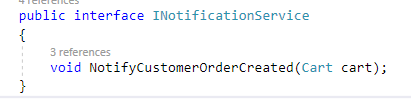
1. הוצאת תלויות לממשקים.
2. 'דחיפת' (או הזרקת) מימושים של הממשקים אל הקלאס (דרך ה ctor)
3. אגב כך, מימוש עקרון SRP על ידי צמצום תחומי האחריות של הקלאסים העושים את הפתרון.

ניקח כדוגמא את

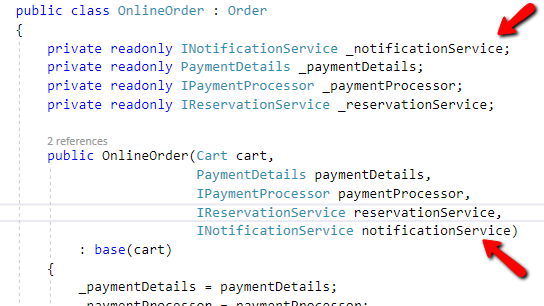


המתודה יושבת בקלאס Cart והמטרה היא המטרה היא להוציאה לממשק חיצוני.

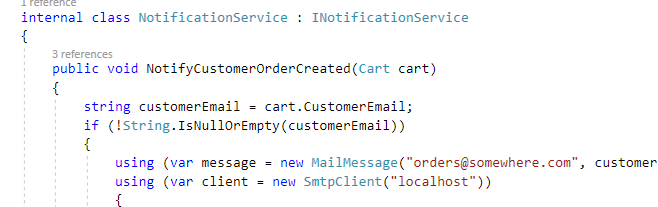
נגדיר ממשק



ניצור קלאס חדש היורש מהקיים שיקבל ממשק זה ב ctor



וניצור מימוש אשר מכיל את אותה מתודה שאותה רצינו להוציא.



מי יוצר מופע של NotificationService ? אפשר באמצעות new. אבל הדרך המקובלת לעשות זאת היא בעזרת מה שנקרא IoC container. זהו מעין פקטורי כללי.

ראה

<http://stackoverflow.com/questions/770039/examples-of-ioc-containers>

על מחשב שאין בו smtp פשוט מייצרים קלאס מדומה



ואותו דוחפים 'מזריקים' אל Order. זה טוב מאד ל unit test

מאמר מוסגר מתוך

Developing Extensible Software - Castro

04. Setting Up the Container Registration-and-Resolve.mp4

שימוש ב ioc-container

ראה את קטע הקוד הבא. שתי המתודות עושות אותה פעולה אבל בשנייה משתמשים ב ioc-container



הקונטיינר הוא לא סתם פקטורי. בבקשת Resolve לא רק שמיוצר מופע של CommAgent אלא גם הוא מגלה ש CommAgent צריך לקבל ב ctor אובייקט המציית ל ICreateSocket ומאחר ויש לו רישום מוקדם מיהו אובייקט זה הוא מאתחל את האובייקט, במקרה זה CreateSocketService ודוחף ('מזריק') אותו ל ctor של CommAgent.

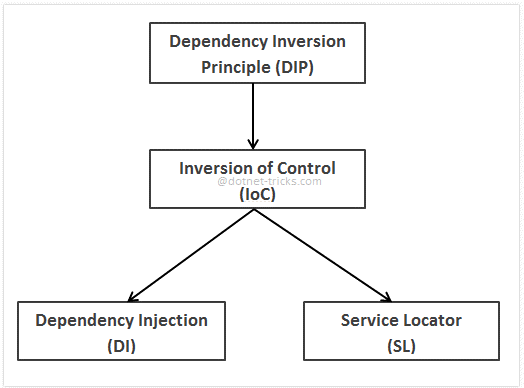
ראה רשימה של ישומים של קונטיינרים

<http://www.dotnettricks.com/learn/dependencyinjection/what-is-ioc-container-or-di-container>

מתוך מאמר

### Understanding Inversion of Control, Dependency Injection and Service Locator

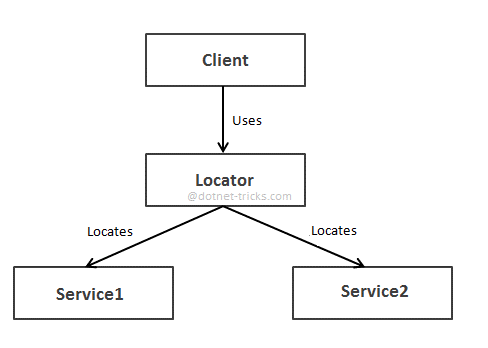
<http://www.dotnettricks.com/learn/dependencyinjection/understanding-inversion-of-control-dependency-injection-and-service-locator>



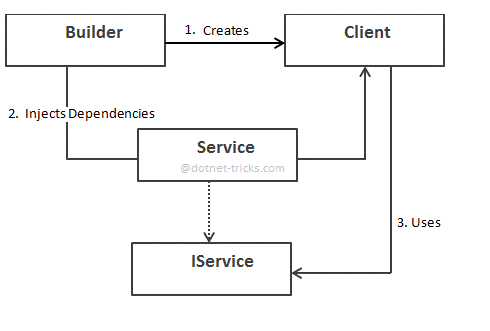
DIP says High level module should not depend on low level module and both should depend on abstraction.

IoC gives some ways to implement DIP

### Using Service locator



### Using Dependency Injection



# Design Patterns Library 2015

## 03 Bridge Pattern

אפליקציה המחזיקה שלשה סוגי מסמכים. "ספר", "מאמר" ו"שאלות נפוצות". לכל אחד מבנה משלו עם מזהים משלו. האפליקציה צריכה לשלוח אותם להדפסה ועל כן כל אחד מהם מציית לממשק משותף ומממש מתודת print משלו.

Interface IMenuScript

{

void print();

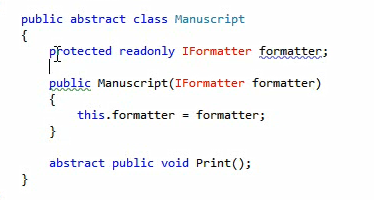
}

אבל, מה אם יש מספר פורמטים של הדפסות ?

פתרון סביר יהיה להגדיר פורמטר. כל סוג קלאס-מסמך יקבל ב ctor את הפורמטר וישתמש בו.

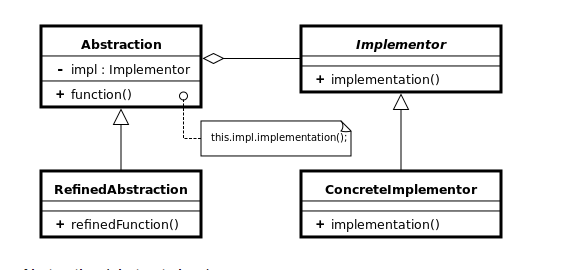
מימושית, כדי להכריח קלאסים להגדיר ctor במבנה מסוים, ובמקרה זה ctor המקבל פורמטר, צריך להשתמש ב abstract class בתור אב משותף במקום בממשק.

זהו קלאס האב האבסטרקטי:



ומכאן ההמשך טריוויאלי. לכל המסמכים יש פורמטר אותו הם מקבלים מבחוץ. כאשר הם מדפיסים הם עושים זאת דרך פורמטר זה והפורמטר מעצב את ההדפסה. נמצא שכל קלאס מממש את היחודיות שלו במתודת print שלו אולם הוא כפוף לאחידות מסוימת דרך הפורמטר.

זו הדיאגרמה המתאימה



כל המסמכים הם בעלי אב משותף, כאן מסומן כ Abstraction

ואב משותף זה פונה אל סדרת קלאסים המממשים תכונה מסוימת, ופניה זו היא דרך קלאס אבסטרקטי Implementor.

## 04 - The Builder Pattern

תצורה זו דנה בבניית קלאס מורכב. בניית הקלאס כרוכה במספר צעדים - לוגיקת הבניה, והוא כמובן מבוסס על פרטים מסוימים שהמשתמש מספק - הנתונים.

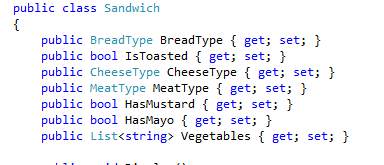
הפתרון הלא טוב הוא לספק קלאס עם מספר רב של ארגומנטים ב ctor, ואם יש קומבינציות שונות, אז יש גם מספר ctor’s. אפשר לחילופין לתת ctor ריק ולאפשר למשתמש לתת את הארגומנטים דרך Prop’s. גם פתרון זה סובל ממספר חסרונות. המשתמש צריך לזכור איזה Prop’s עליו לקבוע, הוא יכול לעשות זאת בסדר שהוא רוצה מה שמסבך את לוגיקת הבנייה, והחיסרון היותר גדול הוא שהמשתמש יכול להשאיר את הקלאס במצב לא חוקי (חסר לו מרכיבים חיוניים) ולנסות להשתמש בו.

נאמר שאדם מזמין סנדוויץ', הוא צריך לפרט את סוג הלחם (שחור לבן) את רמת האפייה שלו, את המילוי המרכזי (סוג הבשר), את רשימת הסלטים שיוכנסו ואת התבלינים. אם שכח מרכיב כלשהוא אז יתכן שהמלצר יעיר לו ויתכן שלא, ואם ינסה לומר מהם התבלינים לפני שאמר מהו המילוי המרכזי המלצר יבקש ממנו להמתין עם בקשה זו לסוף.

תצורת בילדר יוצאת מתוך הנחה שיש מספר הרכבים פופולריים של סנדוויצ'ים ומתמקדת בהם. כאשר יש הרכב מסוים קל יותר לשלוט על אופן הכנת הסנדוויץ' ועל התשאול של המזמין.

התצורה מפרידה בין מרכיבי הסנדוויץ' לבין בניית הסנדוויץ' וקצת מפשטת וממסדת את האופן בה צריך המשתמש להזין את הנתונים.

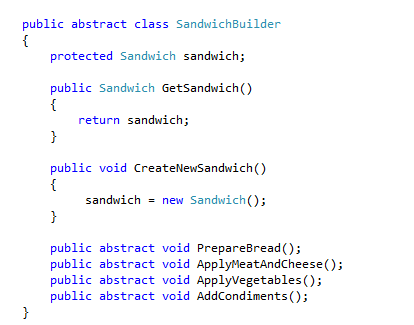
קלאס סנדוויץ' מייצג את מרכיבי הסנדוויץ.



המטרה כאן היא לייצר קלאס כזה.

הקלאס האבסטרקטי SandwichBuilder הוא **אב טיפוס** למי שרוצה ליצור סנדוויץ בהרכב מסוים..

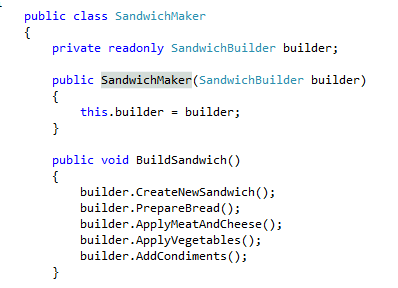
נראה כך



הוא מכריח את מי שמממש אותו להגדיר את כל מרכיבי הסנדוויץ', כלומר את כל המתודות האבסטרקטיות. כך שלא יתכן מצב שיהיה אובייקט סנדוויץ' במצב לא חוקי.

בשלב הבא יוצרים SandwichMaker אשר מגדיר את אופן הכנת הסנדוויץ' בהינתן מימוש של SandwichBuilder .

הוא נראה כך

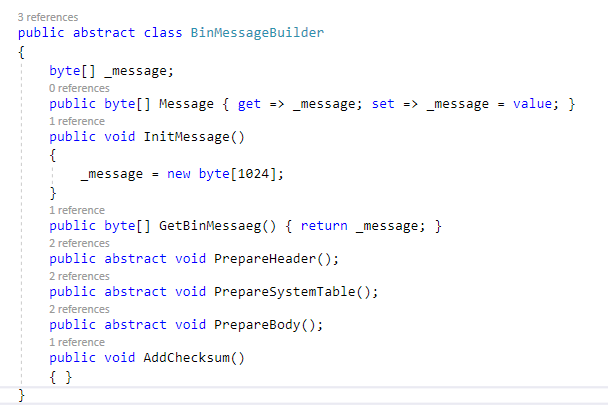


כך יוצרים סנדוויץ'

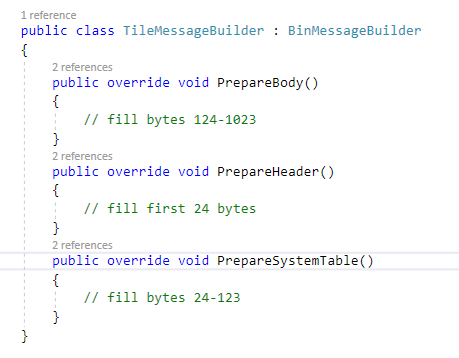


שוב נאמר שרוצים לייצר הודעה בינארית הכוללת מספר שדות. חלק וצריך גם לוגיקה מסוימת כגון דחיסה, הצפנה, קוד תיקון שגיאות וכו.

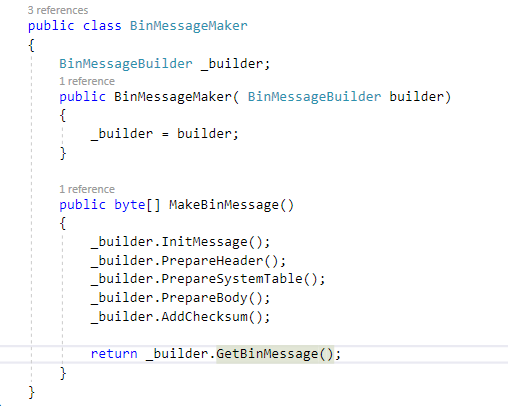
זהו אב טיפוס לכל הודעה בינארית



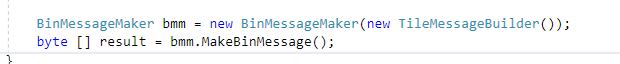
וזהו מימוש מסוים של אב טיפוס זה



קלאס זה יודע, בהינתן בילדר מתאים, לייצר הודעה בינארית.



וזהו אופן השימוש

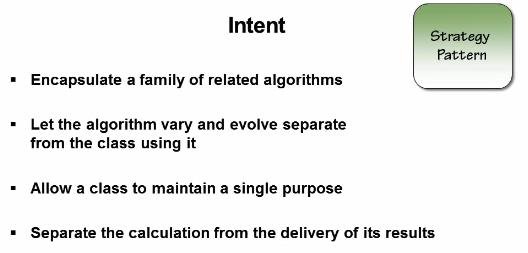


שים לב שהמייקר מוסיף checksum. הוא גם עשוי להוסיף דחיסה וכיו"ב. אין כאן תמיכה בווריאציות של בניה אבל זה נשמע טריוואלי.

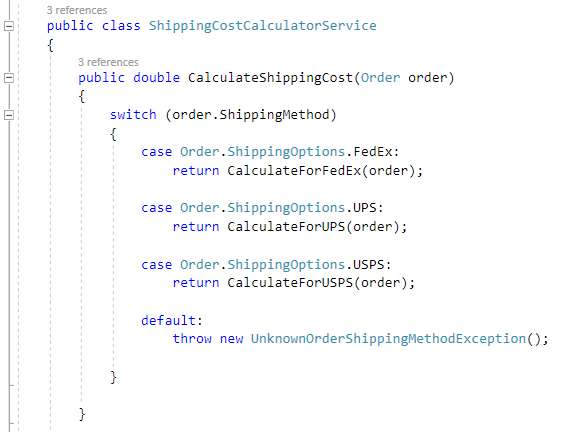
## 28 - Strategy Pattern

נאמר שאתר מכירות מבקש לחשב עלות משלוח. עלות זאת תלויה בחברת המשלוח ולכל חברה אסטרטגיית חישוב משלה.

הכוונה כאן היא מצד אחד לקבץ את כל האסטרטגיות ומצד שני לבודד את ההשפעה של שינוי אסטרטגיה או הוספת אסטרטגיה, מהקלאס הראשי. וכן ליישם את Single Purpose Pattern כך שכל קלאס יהיה אחראי על נושא מצומצם ככל האפשר.



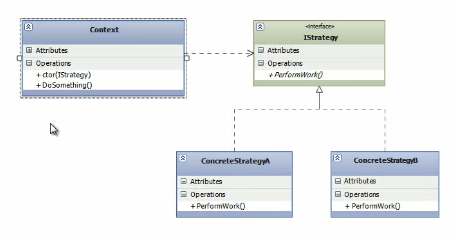
זו הצורה הלא נכונה למימוש:



כללית הצהרות switch הן דגל אדום. יש דרכים טובות יותר לממש שיטות כאלה.

החסרון כאן ברור. המתודה המרכזית נתונה לשינויים על כל שינוי אסטרטגיה מקומי. יש כאן הפרה של OCP ושל SRP.

Strategy Pattern:

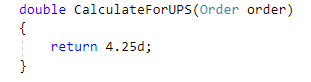


ה context הוא מחשב המשלוחים. הוא מקבל ב ctor שלו את אסטרטגיית המשלוח. הוא בוודאי לא מקבל קלאס ממשי אלא ממשק שמסתיר מאחוריו שיטת חישוב.

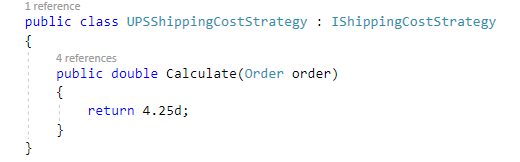
פירוש הדבר שכל שיטת חישוב, שכרגע מיוצגת על ידי מתודה, תיוצג על ידי קלאס ייעודי אשר עונה לממשק אחיד IStrategy. קלאס יעודי זה בוודאי עונה ל SRP בהיותו אחראי לאסטרטגיית חישוב אחת בלבד והמבנה כולו עונה ל OCP באשר שינוי עתידי צפוי להיות מקומי מאד והוספת אסטרטגיה היא טריוויאלית.

פתרון

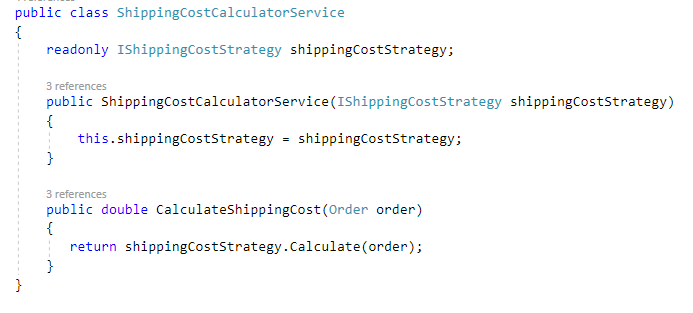
המתודה



הופכת להיות קלאס



והמחשבון שעושה את החישוב עצמו יראה כך



מקבל ב ctor את האסטרטגיה ומיישם אותה בזמן החישוב.

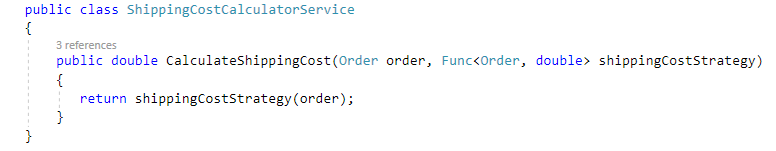
הכנסת אסטרטגיה ל ctor נקראת Strategy injection.

[השאלה כאן היא כיצד מכניסים את אובייקט האסטרטגיה אל ה ctor, הרי בוודאי שלא משתמשים שם ב switch משום שבזה לא הרווחנו כלום, אולי צריך לומר שברגע שהלקוח בוחר שיטת משלוח אז מייצרים את אובייקט האסטרטגיה והוא נכנס בתור שדה ל order]

דרך נוספת לממש את תצורת האסטרטגיה היא באמצעות דליגייט. כלומר, במקום ליצור קלאס בו מונחת אסטרטגיה מסוימת פשוט מייצרים דליגייט. מחשבון המשלוח יקבל דליגייט במקום לקבל ממשק ובאמצעותו יבצע את החישוב.

ואם הגענו עד כאן אין צורך לתת את האסטרטגיה ב CTOR אפשר לתת אותו ישירות למתודה החישוב

[עם כי זה נכון גם בשיטה הראשונה ודומה שזה לא נחשב כ strategy pattern קלאסי]



זהו הדליגייט:



וזו צורת השימוש



[נראה ששימוש בדליגייט יותר פשטני מאשר בקלאס עם מתודה אחת]

## 06 The Command Pattern

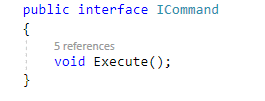
תצורה זו עוסקת בהפיכת פקודה בדידה לאובייקט פקודה. יש מספר יתרונות לתצורה זו ובניהם:

1. הפרדת הקוד המשתמש בפקודה מפרטים הנדרשים לביצוע ומתלויות הנדרשות לביצוע הפקודה
2. מתן אפשרות לביצוע מאוחר של הפקודה. הפקודה נכנסת לתור ומתבצעת בהגיע שעתה.
3. ניתן גם להרחיק את הפקודה למכונה אחרת שתתבצע שם (כמו שיש ב remoting)
4. תמיכה בוולידציה. תמיכה ב undo.

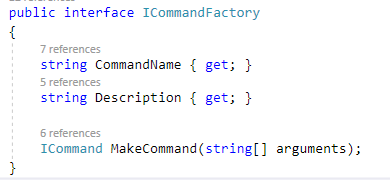
מאחר והפקודה נארזת באובייקט, אין אפשרות לקליינט להכניס ארגומנטים כמו בקריאה רגילה. הארגומנטים צריכים להיארז כחלק מאובייקט הפקודה.

מטבע הדברים צריך גם פקטורי אשר מייצר פקודות בהסתמך על ארגומנטים נתונים. כאן, לצורך הדוגמא, נעשה שימוש בפקטורי דל.

נגדיר ממשק



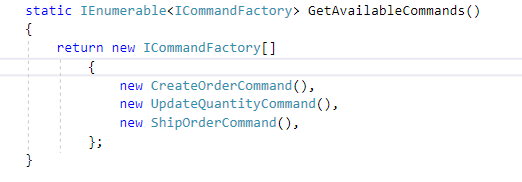
וממשק עבור הפקטורי



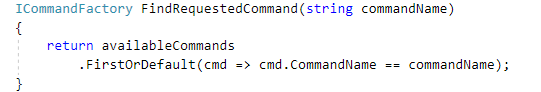
[לא ברור לי למה המחבר בחר להכניס את המטא-דטא ברמת הפקטורי ולא ברמת הפקודה עצמה]

תהליך העבודה הינו כדלקמן:

קבל את רשימה האובייקטים אשר תומכים ב ICommandFactory. דוגמא פשוטה:



מתוכם, מצא את אותו פקטורי שיכול לספק אובייקט-פקודה-מסוים הנדרש לביצוע פקודה מסוימת (בדרך כלל לפי שם)



(במידה ולא נמצא אובייקט מתאים, כדאי להגדיר אובייקט NotFoundCommand ולהחזיר אותו, זה עדיף מלהחזיר ערך null ולהתחיל להתמודד עם המשמעות שלו. יש הקוראים לתצורה זו null command pattern)

יצר את אובייקט הפקודה תוך שימוש בארגומנטים המתאימים. MakeCommand()

קרא למתודה Execute על אובייקט הפקודה (או שמור או העבר את האובייקט לביצוע מאוחר יותר)

## 07 Composite Pattern

הרעיון כאן הוא להתייחס לקבוצה כשם שמתייחסים לפרט.

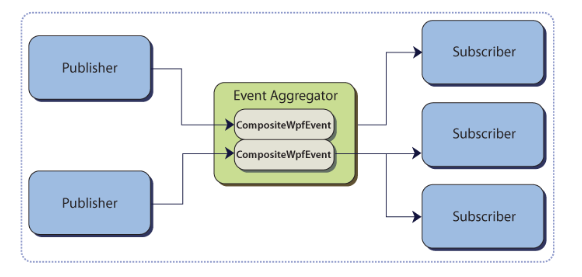
הדוגמא הפשוטה היא שליחת מייל. שולחים מייל ליחיד בדיוק כפי ששולחים מייל לקבוצה. תצורה זו דנה איך לממש מנגנון כזה.

## 09 Event Aggregator

הערה א': זה הרעיון של סקרינר אותו פיתחתי בפרויקט מסוים, כאן כמובן יש תפיסה רחבה יותר וגם תמיכה של ספריות דוט נט.

הערה ב': הווידאו של נושא זה לא כל כך בהיר, מה גם שדוגמת הקוד היא וובית ובנוסף הסטודיו מסרב לפתוח את הדוגמא בטענה של חוסר תאימות. רשימה זו מתבססת על [מאמר](https://msdn.microsoft.com/en-us/library/ff921122.aspx) של MSDN ועל הוידיאו מתוך הקורס של פריזם. Introduction to PRISM\6. Communication\08. Demo Using IEventAggregator.wmv

זו התצורה



המאורעות events המופצים מהמפרסמים למנויים הם מבוססים על קלאסים החייבים לרשת מקלאס בסיס נתון. במקרה של מימוש מיקרוסופטי מדובר על

public class CompositeWpfEvent<TPayload> : EventBase

ובפריזם זה

CompositePresentationEvent<TPayload>

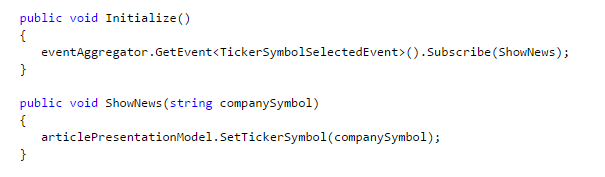
קלאס הפיילוד הוא קלאס נוסף הנושא עמו את **תוכן** המאורע.

לדוגמא

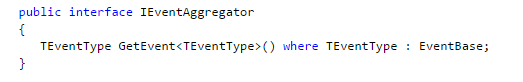


זו הגדרה של מאורע אשר מפרסם עשוי לפרסם, והוא צריך לצרף מחרוזת כאשר הוא מפרסם.

קלאס הרוצה לקבל מאורעות מסוג זה, צריך לתת את הקוד הבא:

כאשר יגיע המאורע תבוצע המתודה ShowNews().

eventAggregator הוא אובייקט מטיפוס **IEvnetAggregator** באשר



(ממשק זה הוא חלק מה Framework, מתכנת לא כותב אותו בעצמו)

פרסום מאורע:



## 10 Facade Pattern

פאסאד בתרגום חופשי זה 'חזית'. לא חזית של מלחמת אלא חזית של בית. מעשית, חזית הבית מסתירה את החדרים ומציגה את דרכי הכניסה והיציאה של הבית.

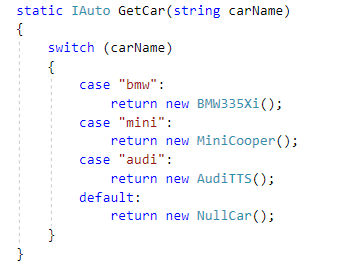
תצורה זו היא טריוויאלית למדי והיא מתוארת כאן רק כדי להבין למה כוונת מתכנתים\מרצים כשהם אומרים פאסאד.

נאמר שיש מספר שירותים לקבלת טמפרטורה בערים שונות בעולם. צריך לפנות אל כל אחד בנפרד ולקבל תשובה ולהציג תשובה זו במקום אחד. אם נאגד את כל הפניות במתודה אחת ונחזיר קלאס אשר מכיל את כל המידע הנדרש, אז נאמר שיש כאן פאסאד.

## 11- Factory Pattern

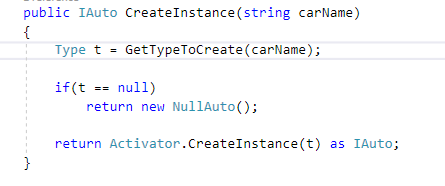
נושא הפקטורי ידוע.

הרי מימוש למפעל מכוניות:



מימוש זה לוקה בכמה חסרונות ואולי הראשון שבהם הוא הפרה של open-close-principle. הוספת מכונית כרוכה בשינוי קלאס הפקטורי. ובכלל קלאס הפקטורי אמור לייצר מכוניות מבלי לדעת בזמן קומפילציה מהם סוגי המכוניות,

נתבונן במימוש הבא

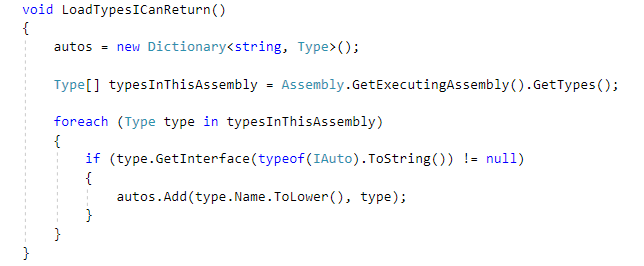


1. ההחלטה איזה טיפוס ליצור, ועצם היצור **עומדים בנפרד**.
2. אין תלות במכונית קונקרטית, רק בממשק.
3. המתודה כלל לא תלויה בסוגי המכוניות וממילא הוספת סוג חדש של מכונית לא משנה אותה.
4. המתודה תומכת גם null-object-pattern

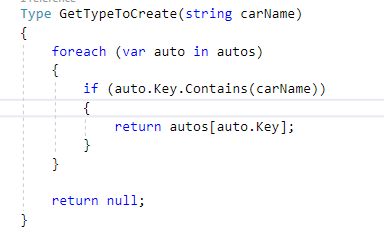
איך יודעים מהם סוגי המכוניות? בעזרת המתודות הבאות.

ב ctor קוראים למתודה הבאה המאכלסת מילון עם שמות המכוניות.

כל קלאס המממש את IAuto נכנס למילון.



על בסיס מילון זה, המתודה הבאה יודעת להחזיר את הקלאס המתאים לשם המכונית:

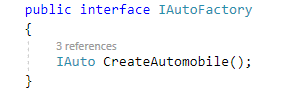


[היה כמובן אפשר להכניס כאן linq קצר יותר]

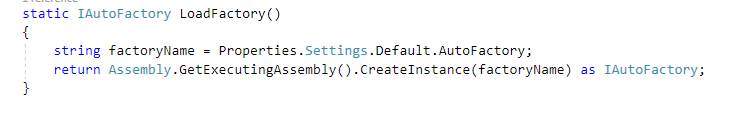
יש ווריאציה חשובה של תצורה זו.

המימוש הנ"ל מניח שיצירת כל סוגי המכוניות אחידה. מאתחלים אובייקט ללא פרמטרים או עם סט פרמטרים קבוע. אבל מה אם יצירת כל סוג מכונית שונה, האתחול שונה, צריך להביא פרמטרים מסוימים מהרשת או לבסס תנאי התחלה מסוימים.

במקרה כזה מוסיפים רובד נוסף. פקטורי ספציפי לכל סוג של מכונית. אותו מימוש מסוים של הפקטורי יודע איך ליצר את המכונית הספציפית לה הוא מיועד. אותו פקטורי ספציפי עונה לממשק



אנו טוענים פקטורי ספציפי של מכונית מסוימת ומייצרים בעזרתו את המכונית.



והשימוש בהתאם הוא



כמובן שבמקרה כזה, אין טעם לטעון את המכוניות עצמם דרך reflection אלא לייצר אותם כרגיל באמצעות new שהרי הפקטורי הספציפי יודע את מי הוא צריך לטעון.

ועכשיו השימוש הוא לא CreateAutomobile אלא CreateSportCar וכן CreateLuxuryCar וכו'.

השלב הבא הוא Abstract Factory

עד עכשיו הנחנו מספר מכוניות, כל אחת מיצרן אחר. למעשה יצרן מייצר מספר סוגי מכוניות, יש לו דגם משפחתי, דגם מיני, דגם ספורט וכו'. כאן אנו מבקשים קלאס אשר ידע ליצר לא רק מכונית אחת אלא סדרה של מכוניות.

מאחר וכבר יש לנו פקטורי ספציפי לכל מכונית, הדרך מכאן קצרה



כל פקטורי ספציפי יצטרך מעכשיו לתמוך בכמה דגמי מכוניות.

State pattern

קצת דומה לאסטרטגיה אלא שאובייקט אסטרטגי הוא בעל התנהגות קבועה בעוד שאובייקט state הוא דינמי. נאמר שיש סרבר העלאת והורדת קבצים ויש לו שלש מצבים: Idle, Ready, Busy, בכל אחד מהמצבים הוא מגיב אחרת לבקשות מהקליינט. לדוגמא במצב Ready הוא יגיב בחיוב לבקשת העלאת קובץ אבל בשאר המצבים הוא יגיב בשלילה, במצב Busy הוא יגיב לבקשת Abort אבל בשאר המצבים הוא יתעלם מהבקשה. נגדיר שלשה אובייקטים ReadyState, IdleState, BusyState וכולם יצייתו לאותו ממשק ICommands. כל אחד מהם יציג התנהגות שונה במענה לבקשה מסוימת.