Visual C# 3.0
סדנת לימוד

Visual Studio 2005/2008
.NET 2.0/3.0/3.5

John Sharp

ייעוץ מקצועי: **מאיר קרודו** dev@krudo.net

את הדוגמאות בספר בגרסת C# 2.0 אפשר להריץ בגרסת C# 3.0, וכל מה שנלמד על Visual Studio 2005 אפשר לבצע ולהריץ בגרסת 2008

לשם שטף הקריאה כתוב ספר זה בלשון זכר בלבד. ספר זה מיועד לגברים ונשים כאחד ואין בכוונתנו להפלות או לפגוע בציבור המשתמשים/ות.

Microsoft Press

הוצאת הוד-עמי

Microsoft Visual C# Step by Step

By **John Sharp**

(כולל תוספות ושינויים)

תרגום: **תומר שפינדל**

עריכה ועיצוב: **שרה עמיהוד**

עיצוב עטיפה: **שרון רז**

תודה ל**יצחק עמיהוד** על הייעוץ המקצועי

www.hod-ami.co.il info@hod-ami.co.il

מהדורה שנייה 2008

של הספר "סדנת לימוד Visual C# 2005"

מסת"ב ISBN 978-965-361-398-0

תוכן עניינים מקוצר

תוכן עניינים

חלק ד' עבודה עם יישומי Windows 347

הקדמה

שפת **Visual C#** של חברת מיקרוסופט היא שפה פשוטה אך חזקה, אשר מיועדת עבור מפתחים המעוניינים לבנות יישומים אשר מתבססים על **Microsoft .NET Framework**. שפה זו משמרת חלק גדול מהיתרונות של השפות ++C ושל Visual Basic, ובאותו זמן פותרת חלק גדול מבעיות חוסר העקביות שלהן וכתוצאה - התקבלה שפה נקייה ולוגית יותר. ל- C# 2.0 נוספו מספר אלמנטים חדשים וביניהם Generics (גנריות), Iterators (חיזרורים, איטרטורים) וגם Anonymous methods (שיטות אנונימיות), ולגרסה האחרונה **C# 3.0** נוספו אלמנטים נוספים כמו מאפיינים אוטומטיים, אתחול אובייקטים ואוספים, שיטות הרחבה, סוגים אנונימיים וביטוי למבדא. שיפורים אלה בשפה נוצרו כבסיס לתמיכה ב-**LINQ** שהינו המאפיין המרכזי שקיבלנו בגרסה האחרונה. סביבת הפיתוח של **Visual Studio 2005/2008** מאפשרת סביבת עבודה שבאמצעותה ניתן להפעיל וליישם את החידושים בקלות ונוחות. היא גם מקלה על השימוש בכלים החדשים. האשפים והשיפורים הרבים שנוספו לסביבת הפיתוח יכולים לתרום רבות לרמות התפוקה של מפתח יישומים.

מטרת הספר ללמד את יסודות התכנות בשפת C# באמצעות Visual Studio 2005/2008 (**הדוגמאות בספר בגרסת C# 2.0 הם בסביבת 2005 של Visual Studio והדוגמאות בגרסת C# 3.0 הם בסביבת 2008 של Visual Studio, אבל ניתן לבצע ולהריץ את כל הדוגמאות בגרסת 2008 של Visual Studio**) וסביבת העבודה NET Framework. תלמד על הכלים השונים של שפת C# וכיצד להשתמש בהם לבניית יישומים שפועלים בסביבת מערכת ההפעלה של Windows. במהלך הלימוד תרכוש הבנה עמוקה של C# ותיישם אותה לבניית יישומי Windows Forms (עולם הטפסים), כדי לגשת אל מסדי הנתונים של Microsoft SQL Server, לפיתוח יישומי רשת של ASP.NET ולבנייה של שירותי אינטרנט ושימוש בהם.

על גרסת ה- .NET framework 2.0, 3.0 ו-3.5

בשנת 2002 הוכרזה והופצה טכנולוגיית NET framework. על ידי חברת מיקרוסופט, בגרסתה הראשונה (**גרסה 1.0**). NET framework. הכילה מספר שפות פיתוח חדשות, וטכנולוגיית פיתוח עבור דפי אינטרנט שנקרא ASP.NET. יחד עם NET.,. החלה מיקרוסופט בשיווק סביבת הפיתוח הייעודית, שנקראה **Visual Studio.NET**, או בשמה הפחות נפוץ, VS 7.

בעקבות מספר באגים ובעיות, מיהרה מיקרוסופט להוציא, בתחילת 2003, גרסה משודרגת של NET.,. והיא קיבלה את מספר **הגרסה 1.1**. הגרסה החדשה הכילה בעיקר תיקוני באגים ומספר עדכוני תוכנה.

במהלך השליש האחרון של שנת 2005 הוכרזה על ידי מיקרוסופט הגרסה השנייה ל- NET. Framework. (**גרסה 2.0**), ויחד עימה הופץ מנוע NET. משודרג, חבילת טכנולוגיות חדשה, וגם סביבת פיתוח חדשה בשם **Visual Studio 2005** (או בשם הגרסה שלה, VS 8). הגרסה החדשה הוסיפה מחלקות קוד חדשות רבות, והציגה שיפורי תוכנה ושיפורי אבטחה רבים.

במהלך דצמבר 2006, הוכרזה גרסת שדרוג ושיפור נוספת ל-NET., שפותחה כחלק מפיתוח מערכת ההפעלה Windows Vista. הגרסה החדשה (**גרסה 3.0**) פותחה כתוספת ל- NET 2.0., ואינה מחליפה אותה, למעשה היא משלבת את הכח של מנוע הריצה 2.0 עם מספר טכנולוגיות חדשות (WPF, WCF, WF). לפירוט נוסף בחן את הכתובת הבאה:

http://msdn2.microsoft.com/en-us/library/aa480198.aspx

במהלך נובמבר 2007 הוכרזה הגרסה של ה-Framework **גרסה 3.5** הכוללת בתוכה את C# 3.0. כמו כן שוחררה סביבת הפיתוח Visual Studio 2008. ניתן להוריד את הגרסה החינמית של סביבת הפיתוח שנקראת Express Edition, כולל קבצי עזרה (MSDN), בכתובת הבאה:

http://msdn.microsoft.com/en-us/vstudio/products/aa700831.aspx

ניתן ליישם ולהריץ את דוגמאות המקור עם הגרסה האחרונה של סביבת הפיתוח Visual Studio 2008.

מציאת נקודת הפתיחה האידיאלית עבורך

ספר זה נועד לסייע לך לפתח יכולות במספר תחומים עיקריים. הוא מיועד גם לחסרי ניסיון בתחום התכנות וגם לאלה שמגיעים מרקע תכנות בשפות אחרות, כגון C, C++, Java או Visual Basic. כדי למצוא את נקודת הפתיחה האידיאלית עבורך, תוכל להיעזר בטבלה הבאה:

רמת הידע שלך	השלבים שעליך לבצע
אם אינך מכיר את עקרונות התכנות מוכוון/מונחה העצמים (object oriented programming)	1. התקן את קבצי התרגול כפי שמוסבר בסעיף "התקנת קבצי התרגול ושימוש בהם". 2. קרא את הפרקים שבחלקים א', ב' ו-ג' של הספר. 3. קרא את חלקי הספר ד', ה' ו-ו' בהתאם לרמת הניסיון וההתעניינות שלך.
אם לא עסקת בתכנות בשפת C#.	1. התקן את קבצי התרגול כפי שמוסבר בסעיף "התקנת קבצי התרגול ושימוש בהם". עבור במהירות על חמשת הפרקים הראשונים, כדי ללמוד על C# ועל Visual Studio 2005, ולאחר מכן התמקד בעיקר בפרקים 6 עד 19. 2. קרא את חלקי הספר ד', ה' ו-ו' בהתאם לרמת הניסיון וההתעניינות שלך.
אם אתה מגיע מרקע תכנות בשפת C, C++ או Java.	1. התקן את קבצי התרגול כפי שמוסבר בסעיף "התקנת קבצי התרגול ושימוש בהם". 2. עבור במהירות על שבעת הפרקים הראשונים, כדי ללמוד על C# ועל Visual Studio 2005, ולאחר מכן התמקד בעיקר בפרקים 8 עד 19. 3. כדי ללמוד על בניית יישומי Windows ושימוש במסדי נתונים, קרא את החלקים ד' ו-ה' בספר. 4. כדי ללמוד על בניית יישומי רשת ושירותי רשת, קרא חלק ו'.

אם אתה מגיע מרקע תכנות בשפת Visual Basic 6.	1. התקן את קבצי התרגול כפי שמוסבר בסעיף "התקנת קבצי התרגול ושימוש בהם". 2. למד את החסר לך מתוך חלקי הספר א', ב' ו-ג'. 3. כדי ללמוד על בניית יישומי Windows קרא את חלק ד'. 4. כדי ללמוד על גישה למסדי נתונים קרא את חלק ה'. 5. כדי ללמוד על בניית יישומי רשת ושירותי רשת, קרא חלק ו'. 6. קרא את הטבלאות המסכמות אשר מופיעות בסוף כל פרק, כדי ללמוד על מבנים ספציפיים של C# ושל Visual Studio 2005.
אם אתה מעיין בספר לאחר שכבר סיימת את התרגילים.	1. היעזר באינדקס או בתוכן העניינים כדי למצוא מידע על נושאים שמעניינים או דרושים לך. 2. קרא את הטבלאות אשר מופיעות בסוף כל פרק, כדי לקרוא סקירה קצרה של התחביר והטכניקות המפורטים באותו פרק.

מוסכמות ומאפיינים בספר

ספר זה מציג את המידע באמצעות מוסכמות שמטרתן העברת המידע באופן ברור וקריא. לפני שתתחיל ללמוד בספר, קרא את הרשימה הבאה אשר מפרטת מוסכמות שהשתדלנו לעקוב אחריהן במהלך כתיבת הספר, ומאפייני כתיבה ותכנות אחרים שעשויים לשמש אותך במהלך הלימוד.

מוסכמות

- כל תרגיל הוא סדרה של פעולות. כל פעולה ממוספרת במספר סידורי (1, 2 וכן הלאה).
- הערות "טיפ" מכילות מידע נוסף או דרך חלופית לבצע פעולה כלשהי.
- הערות "חשוב" מעבירות לך מידע שעליך לבדוק לפני שתמשיך.
- כאשר מופיע סימן פלוס (+) בין שני מקשים, עליך ללחוץ עליהם בו-זמנית. לדוגמה, אם כתוב "הקש Alt+Tab" - עליך ללחוץ על מקש Tab בעודך מחזיק את מקש Alt. לחץ תחילה על המקש הראשון משמאל.

מאפיינים נוספים

- במסגרות המופיעות בספר תוכל למצוא הרחבה והסבר של תרגילים שונים. המסגרות עשויות להכיל נתוני רקע, עצות עיצוב או כל דבר אחר הקשור לנושא התרגיל.
- בסופו של כל פרק תמצא טבלת סיכום. תוכל להשתמש בה כדי לסכם את מה שלמדת בפרק, וגם כתזכורת כיצד לבצע את הפעולות השונות הרלוונטיות לכל פרק.

תוכן מקוון נלווה

אתר תוכן מקוון מכיל תכנים וקישורים עבור ספר זה (מקור באנגלית):

http://www.microsoft.com/mspress/companion/0-7356-2129-2/

עדכונים טכנולוגיים

בעוד הטכנולוגיות מתעדכנות, יתווספו לדף האינטרנט Technology Updates - קישורים אשר מובילים למידע נוסף בנושא. בקר מדי פעם בדף זה, כדי להתעדכן על סביבת הפיתוח Visual Studio 2005/2008 ועל טכנולוגיות אחרות:

http://www.microsoft.com/mspress/updates/

דרישות תוכנה

כדי להשלים את כל התרגילים בספר זה, תזדקק לחומרה ולתוכנה הבאים:

- Microsoft Windows XP Professional Edition with Service Pack 2, Microsoft Windows Server 2003 with Service Pack 1, or Windows 2000 with Service Pack 4. (Microsoft Windows 2000 Datacenter Server is not supported.)
- Microsoft Visual Studio 2005/2008 Standard or Professional Edition, including SQL Server 2005/2008 Express.
- 766 MHz Pentium or compatible processor (1.5 GHz Pentium recommended).
- 256 MB RAM (512 MB or more recommended).
- Video monitor (800 × 600 or higher resolution) with at least 256 colors (1024 × 768 High Color 16-bit recommended).
- CD-ROM or DVD-ROM drive.
- Microsoft Mouse or compatible pointing device.

כמו כן, תצטרך כניסת Administrator למחשב שלך כדי לעצב את SQL Server 2005 Express Edition וכדי לשנות את Windows Registry בפרק 28.

שים לב: התוכנה Visual Studio 2005/2008 אינה חלק מספר זה! הקבצים שבאתר ההוצאה מכילים את דוגמאות הקוד הדרושות לביצוע התרגילים.

את Visual Studio 2005/2008 חייבים לרכוש בנפרד או להוריד גרסה חינמית מאתר האינטרנט:

http://msdn.microsoft.com/en-us/vstudio/products/aa700831.aspx

התקנת קבצי התרגול ושימוש בהם

באתר הוד-עמי תמצא את קבצי התרגול (קוד המקור) בהם עליך להשתמש בעת ביצוע התרגילים שבפרקי הספר השונים. מטרת קבצי התרגול לחסוך ממך יצירת קבצים שאינם דרושים לתרגיל, ובעיקר - טרחה להקליד פקודות ושמות ארוכים. הקבצים וההוראות שבתרגיל מאפשרים ללמוד על ידי עשייה - זו דרך טובה ויעילה לרכוש מיומנויות חדשות, להטמיע ולפתח אותן.

התקנת קבצי התרגול

כדי להשתמש בקבצי התרגול לביצוע התרגילים, עליך להתקין אותם בדיסק שבמחשב שלך:

1. קבצי התרגול (קוד המקור) נמצאים באתר הוד-עמי www.hod-ami.co.il. היכנס לאתר, מצא את הלינק **קוד מקור** תחת ספר זה והורד את הקובץ למחשב. פתח את קובץ ה-ZIP לתיקייה חדשה. פתח את המחשב שלי (My Computer) שעל שולחן העבודה, עבור לתיקיה שפתחת ולחץ לחיצה כפולה על הקובץ **StartCD.exe**. על המסך יוצגו תנאי השימוש באופן אוטומטי. אשר.
2. כעת יופיע תפריט שמכיל אפשרויות שונות.
3. בחר **Install Practice Files**.
4. פעל על פי ההוראות שיוצגו במסך. קבצי התרגול יותקנו בתיקייה זו:

My Documents\Microsoft Press\Visual CSharp Step by Step

5. **להתקנת קבצי התרגול של חלק ז' C# 3.0:**

לחץ על הלינק **קוד מקור1**, לחץ על save (שמור), שמור במקום כלשהו בדיסק ואז פתח את הקובץ לאותה תיקייה בה נמצאים שאר הקבצים (ראה סעיף 4).

קביעת התצורה של SQL Server Express Edition

בתרגילים שבחלק ה' של הספר עליך לגשת לשרת SQL Server Express Edition כדי ליצור את מסד הנתונים Northwind. אם אתה משתמש בשרת SQL Server 2005 Express Edition, עליך להיכנס למערכת ההפעלה של המחשב שלך בתור מנהל מערכת (Administrator), ולבצע את הפעולות הבאות, כדי שתוכל לגשת לחשבון המשתמש אשר ישמש אותך לביצוע התרגילים.

1. בסביבת Windows, פתח את תפריט **התחל (Start)** ← **כל התוכניות (Programs All)** ← **עזרים (Accessories)** ובחר **Command Prompt** כדי לפתוח את חלון הפקודות.
2. בחלון הפקודות, הקלד את הפקודה הבאה: sqlcmd –S *YourServer*\SQLExpress –E

במקום YourServer כתוב את שם המחשב שלך. אם אינך יודע את שם המחשב, הפעל את הפקודה hostname לפני הפעלת הפקודה sqlcmd.

3. לאחר prompt 1> הקלד את הפקודה הבאה עם הסוגריים המרובעים, והקש Enter:

sp_grantlogin [YourServer\UserName]

במקום YourServer כתוב את שם המחשב שלך ובמקום UserName כתוב את שם המשתמש בחשבון הנוכחי.

4. לאחר prompt 2> הקלד את הפקודה הבאה והקש Enter: go

אם תופיע הודעת שגיאה, ודא שהקלדת את הפקודה sp_grantlogin ללא טעויות, כולל הסוגריים המרובעים והנתונים שאחריה.

5. לאחר prompt 1> הקלד את הפקודה הבאה ואת הסוגריים המרובעים, והקש Enter:

sp_addsrvrolemember [YourServer\UserName], dbcreator

6. לאחר prompt 2> הקלד את הפקודה הבאה והקש Enter:

```
go
```

אם תופיע הודעת שגיאה, ודא שהקלדת את הפקודה sp_addsrvrolemember ללא טעויות, כולל הסוגריים המרובעים והנתונים שאחריה.

7. לאחר prompt 1> הקלד את הפקודה הבאה, כולל הסוגריים המרובעים, והקש Enter:

```
exit
```

8. סגור את חלון הפקודות.

שימוש בקבצי התרגול

בכל אחד מהפרקים שבספר זה מפורט בדיוק כיצד להשתמש בקבצי התרגול אשר דרושים בו. כאשר עליך להשתמש בקובץ תרגול, יוסבר לך כיצד עליך לפתוח את הקובץ. התרגילים שבפרקים השונים מדמים פרויקטים אמיתיים, כדי שמאוחר יותר תוכל ליישם בקלות בעבודתך את הדברים שלמדת. בטבלה הבאה מפורטים הפרויקטים השונים שבקבצי התרגול והתיאורים שלהם.

פרויקט	תיאור
פרק 1	
TextHello	פרויקט זה נועד לסייע לך להתחיל. מטרתו לעבור על שלבי יצירת תוכנית פשוטה אשר מציגה טקסט על המסך.
WinFormHello	פרויקט זה מציג את הטקסט בחלון על ידי שימוש ב- Windows Forms.
פרק 2	
PrimitiveDataTypes	פרויקט זה מדגים כיצד יש להכריז על משתנים הפרימיטיביים מכל הסוגים, כיצד להציב ערכים במשתנים אלה וכיצד להציג את ערכיהם בחלון.
MathsOperators	תוכנית זו מדגימה את השימוש באופרטורים האריתמטיים (+ - * / %).
פרק 3	
Methods	בפרויקט זה עליך לבחון מחדש את הקוד שבפרויקט הקודם, כדי להבין כיצד הוא מורכב באמצעות שיטות (methods).
DailyRate	פרויקט זה נועד להנחות אותך בכתיבת שיטות משלך (באופן ידני או בעזרת אשף), בהפעלת שיטות, ובפסיעה דרך (stepping through) הקריאות לשיטות באמצעות המנפה (debugger) של Visual Studio.
פרק 4	
Selection	פרויקט זה מראה כיצד יש להשתמש במשפטי if מקוננים כדי להשוות בין שני תאריכים.
switchStatement	תוכנית פשוטה זו מפעילה משפט switch כדי להמיר תווים לפורמט XML.
פרק 5	
Iteration	פרויקט זה כולל מקטעי קוד עבור כל אחד ממשפטי האיטראציה, ומציג את הפלט שמתקבל מכל אחד מהם.
whileStatment	פרויקט זה מפעיל משפטי while כדי לקרוא תוכן של קובץ מקור שורה אחר שורה, ולהציג את השורות בתיבת טקסט של Windows.

פרויקט	תיאור
doStatment	פרויקט זה מפעיל משפטי do כדי להמיר מספר למחרוזת המייצגת אותו.
פרק 6	
MathsOperators	פרויקט זה בוחן שוב את הפרויקט MathsOperators מפרק 2 ומכשיל את פעולת התוכנית באמצעות חריגים בלתי מטופלים. על ידי הוספת מילות המפתח try ו-catch התוכנית הופכת ליציבה יותר ואינה נכשלת עוד.
פרק 7	
Classes	פרויקט זה עוסק בעקרונות להגדרת מחלקות, הכוללות בנאים ציבוריים (public constructors), שיטות (methods) ושדות פרטיים (private fields). הוא עוסק גם ביצירת מופעים של מחלקות באמצעות מילת המפתח new, שיטות סטטיות (static) ושדות.
פרק 8	
Parameters	תוכנית זו בוחנת את ההבדלים בין פרמטרים של ערך (value parameters) ופרמטרים של הפניה (parameters reference). התוכנית מדגימה את השימוש במילות המפתח ref ו-out.
פרק 9	
StructsAndEnums	פרויקט זה מפעיל את סוג הנתונים enum לייצוג ארבעת הסימנים של קלפי המשחק, ובסוג struct - לייצוג התאריך.
פרק 10	
Aggregates	פרויקט זה מפעיל את הפרויקט הקודם כדי ליצור מחלקת אוסף (collection class) בשם ArrayList, המאגדת יחדיו מספר קלפי משחק.
פרק 11	
ParamsArrays	פרויקט זה ממחיש את השימוש במילת המפתח params ליצירת שיטה אשר יכולה לקבל מספר בלתי מוגבל של פרמטרים מסוג int ולהחזיר את הערך הנמוך ביותר מביניהם.
פרק 12	
CSharp	פרויקט זה בונה היררכיה של ממשקים ומחלקות כדי לדמות קריאה של קובץ מקור של C# וסיווג תכולתו על פי מטבעות (tokens) שהם מזהים (identifiers), מילות מפתח (keywords), אופרטורים ועוד. בתור דוגמה, הפרויקט גוזר מחלקות מהממשק המרכזי שמטרתן הצגת המטבעות בתחביר צבעוני בתיבת טקסט עשיר.
פרק 13	
UsingStatment	פרויקט זה ממחיש שמקטע של קוד מהפרק הקודם אינו עמיד בפני חריגים. הוא מראה באמצעות משפט using כיצד לגרום לכך שהקוד יהיה עמיד בפני חריגים.
פרק 14	
Properties	פרויקט זה מציג יישום פשוט של Windows אשר בעזרת מספר מאפיינים (properties) הוא מציג באופן מתמשך את גודל החלון המרכזי של היישום.

פרויקט	תיאור
פרק 15	
Indexers	פרויקט זה מפעיל שני סדרנים (indexers): אחד לאיתור מספר טלפון לפי שם, ואחד לאיתור שם לפי מספר טלפון.
פרק 16	
Delegates	פרויקט זה מציג את השעה בפורמט דיגיטלי באמצעות נציגים (delegates). לאחר מכן הקוד מופשט על ידי שימוש באירועים (events).
פרק 17	
BinaryTree	פרויקט זה מראה כיצד להשתמש בעיקרון הגנריות (Generics), כדי לממש מבנה אשר יכול להכיל כל סוג של אלמנט.
BuildTree	פרויקט זה מדגים את השימוש בעיקרון הגנריות למימוש שיטה אשר יכולה לקבל כל סוג של פרמטר.
פרק 18	
BinaryTree	פרויקט זה מראה כיצד לממש את הממשק הגנרי IEnumerator<T> כדי ליצור מונה (enumerator) עבור המחלקה הגנרית BinaryTree.
IteratorBinaryTree	פרויקט זה מפעיל איטרציה (Iterator) כדי לחולל מונה עבור המחלקה הגנרית BinaryTree.
פרק 19	
Operators	פרויקט זה בונה שלושה מבנים (structs) בשמות Hour, Minute ו-Second המכילים אופרטורים המוגדרים על ידי המשתמש. קוד זה מפושט על ידי שימוש באופרטור ההמרה (conversion operator).
פרק 20	
BellRingers	פרויקט זה הינו יישום של טפסים המדגים שימוש בסיסי בפקדי Windows Forms.
פרק 21	
BellRingers	פרויקט זה הינו הרחבה לפרויקט מפרק 20 ונוספים בו תפריטי קיצור ותפריטים נפתחים בממשק המשתמש.
פרק 22	
CustomerDetails	פרויקט זה ממחיש, באמצעות דוגמה של נתוני לקוח, את ביצוע בדיקות תקינות לקלט המתקבל מהמשתמש.
פרק 23	
DisplayProducts	פרויקט זה מראה כיצד להשתמש במודל ADO.NET כדי להתחבר למסד הנתונים Northwind, ולשלוף נתונים מהטבלה Products. פרויקט זה מפעיל את האשף Data Source Configuration כדי לחולל מקור נתונים שבאמצעותו ניתן להתחבר למסד הנתונים. הוא גם מפעיל אובייקטים מסוג DataSet, DataTable ו-DataAdapter שבאמצעותם ניתן לכרוך (bind) את מקור הנתונים לפקד DataGridView. הפקד הזה משמש להצגת הנתונים בטופס Windows.

פרויקט	תיאור
ReportOrders	פרויקט זה מראה כיצד לגשת למסד נתונים על ידי כתיבת קוד של ADO.NET, ולא לעשות זאת על ידי האשף Source Configuration Data. יישום זה שולף נתונים מהטבלה Orders שבמסד הנתונים Northwind Traders.
פרק 24	
ProductsMaintenance	פרויקט זה מדגים שימוש באובייקטים מסוג DataSet, DataTable ו-DataAdapter לעדכון הנתונים שבמסד הנתונים. יישום זה מפעיל את הפקד DataGridView במסגרת טופס Windows ומאפשר למשתמש לערוך את הנתונים בטבלה Products שבמסד הנתונים Northwind.
פרק 25	
HonestJohn	פרויקט זה יוצר אתר אינטרנט פשוט על פי המודל ASP.NET. האתר מאפשר למשתמש להזין נתונים של עובדים בחברת פיתוח תוכנה דמיונית.
פרק 26	
HonestJohn	פרויקט זה הינו הרחבה של הפרויקט מהפרק הקודם. הוא מדגים כיצד לערוך בדיקת תקינות לקלט המתקבל מהמשתמש ביישום רשת של ASP.NET.
פרק 27	
Northwind	פרויקט זה מדגים כיצד להשתמש באבטחה במסגרת הטופס (Form-based security) לאימות זהות המשתמש. יישום זה גם מדגים כיצד להשתמש ב-ADO.NET מתוך טופס Web של ASP.NET, ומראה כיצד לחקור ולעדכן מסדי נתונים באופן יעיל.
פרק 28	
NorthwindServices	פרויקט זה הינו יישום של שירות אינטרנט (Web Service), אשר מאפשר גישה מרחוק באמצעות האינטרנט, לנתונים שבטבלה Products שנמצאת במסד הנתונים Northwind.
ProductInfo	פרויקט זה מציג כיצד ליצור יישום Windows שצורך שירות הרשת. הוא גם מראה כיצד לקרוא לשיטות שנמצאות בשירות NorthwindServices.
פרק 29	
Intro C_Sharp 3	פרויקט זה מציג את המאפיינים החדשים שנוספו לגרסת C# 3.0.
פרק 30	
Intro to LINQ	פרויקט זה מכיל דוגמאות מבוא לשימוש בטכנולוגיית LINQ.
פרק 31	
LINQ to SQL	פרויקט זה מציג את היכולות של LINQ to SQL לביצוע שאילתות מול בסיס הנתונים.

פרויקט	תיאור
פרק 32	
LINQ to XML	פרויקט זה מציג את היכולות של LINQ to XML לביצוע שאילתות ובנייה של XML.

בנוסף לפרויקטים שמפורטים בטבלה, יש בפרקי הספר פרויקטים נוספים שפתרונותיהם מאוחסנים בתיקייה של הפרק בקבצי התרגול, ומסומנים במילה Complete.

הסרת קבצי התרגול

כדי למחוק את קבצי התרגול מהמחשב שלך, עליך לנהוג כך:

1. **בלוח הבקרה** (Control Panel) לחץ לחיצה כפולה על **הסרה והוספה של תוכניות** (Add or Remove Programs).
2. בחר מרשימת התוכניות המותקנות את **Microsoft Visual C# 2005 Step By Step**.
3. לחץ **Remove**.
4. עקוב אחר ההוראות שעל המסך, כדי להסיר את קבצי התרגול.

תמיכה ללומד בספר

נעשו כל המאמצים כדי להבטיח שספר זה וקבצי התרגול יהיו מדויקים ככל הניתן. תיקונים ושינויים עבור הספר המקורי באנגלית יפורסמו באתר Microsoft Knowledge Base. כדי לראות את רשימת התיקונים עבור ספר זה עליך לבקר באתר הבא:

http://support.microsoft.com/kb/905035/

בתרגום לעברית בספר זה השתדלנו להכניס את כל התיקונים שהיו באתר בזמן התרגום.

הוצאת הוד-עמי תוכל לסייע לך בהורדת הקבצים מהאתר, או בכל תקלה שתמצא בספר. **הצאת הוד-עמי אינה נותנת שירותי סיוע בתכנות או בהבנת הכתוב.** לשם כך יש לפנות אל יועץ תכנות.

הדרך לפניות אל הוד עמי: info@hod-ami.co.il

שאלות והערות

אם יש לך הערות, שאלות או רעיונות בנוגע לספר או בנוגע לקבצי התרגול הנלווים, או אם יש לך שאלות שלא נפתרו על ידי ביקור באתרי התמיכה, שלח אותם בדואר אלקטרוני אל הוצאת Microsoft:

mspinput@microsoft.com

חלק א'

היכרות עם Microsodt Visual C#
Microsoft Visual Studio 2005 –1

בחלק זה:

את הדוגמאות בספר בגרסת C# 2.0 אפשר להריץ בגרסת C# 3.0, וכל מה שנלמד על Visual Studio 2005 אפשר לבצע ולהריץ בגרסת 2008

פרק 1

ברוך הבא ל-#C

בסיום פרק זה, תוכל:

- להשתמש בסביבת הפיתוח של Visual Studio 2005.
- ליצור יישום מסוף (console application) של #C.
- להשתמש במרחבי שמות (namespaces).
- ליצור יישום Windows Forms של #C.

Microsoft Visual C# הינה שפה בעלת עוצמה רבה, מונחית-רכיבים ומוכוונת עצמים (Object-Oriented) של Microsoft. לשפה #C יש תפקיד חשוב בעיצוב של Microsoft .NET Framework, והדבר מזכיר את התפקיד שהיה ל-C בפיתוח של UNIX. אם עבדת בעבר עם תוכנות כגון C, ++C או Java, התחביר (syntax) של #C ייראה לך מוכר, כי גם בו יש שימוש בסוגריים מסולסלים ({}) להכלת בלוקים של קוד. אבל גם אם אינך רגיל לתכנת בשפות אחרות, תוכל להשתלט במהרה על התחביר של #C. כל שעליך ללמוד, זה להציב את הסוגריים המסולסלים ואת הנקודה-פסיק (;) במקומות הנכונים. ספר זה ידריך אותך לעשות זאת.

בחלק הראשון של הספר תלמד את היסודות של #C. תגלה כיצד להכריז על משתנים (variables) וכיצד להשתמש באופרטורים (operators) כגון פלוס (+) ומינוס (-), כדי ליצור ערכים (values). תלמד כיצד לכתוב שיטות (methods) ולהעביר ארגומנטים (arguments) לשיטות. גם תלמד כיצד להשתמש במשפטי בחירה (selection statements) כגון **if** ובמשפטי לולאה (iteration statements) כגון **while**. לאחר כל זאת תבין כיצד #C מפעילה את החריגים (exceptions) כדי להתמודד עם שגיאות באופן אלגנטי וידידותי למשתמש. נושאים אלה מהווים את הגרעין של שפת #C, ומבסיס זה תתקדם לנושאים מורכבים יותר בהמשך פרקי הספר.

תכנות בסביבת Visual Studio 2005

Visual Studio 2005 היא סביבת פיתוח עשירה, המכילה את כל הכלים שתזדקק להם כדי ליצור פרויקטים, קטנים וגדולים כאחד בשפת #C. באפשרותך אפילו ליצור פרויקטים אשר משלבים באופן רציף מודולים (modules) משפות תכנות אחרות. בתרגיל הראשון תפעיל את סביבת הפיתוח של Visual Studio 2005 ותלמד כיצד ליצור יישום מסוף (console), אשר אין לו ממשק גרפי והפקודות מבוצעות משורת הפקודה (Command Prompt).

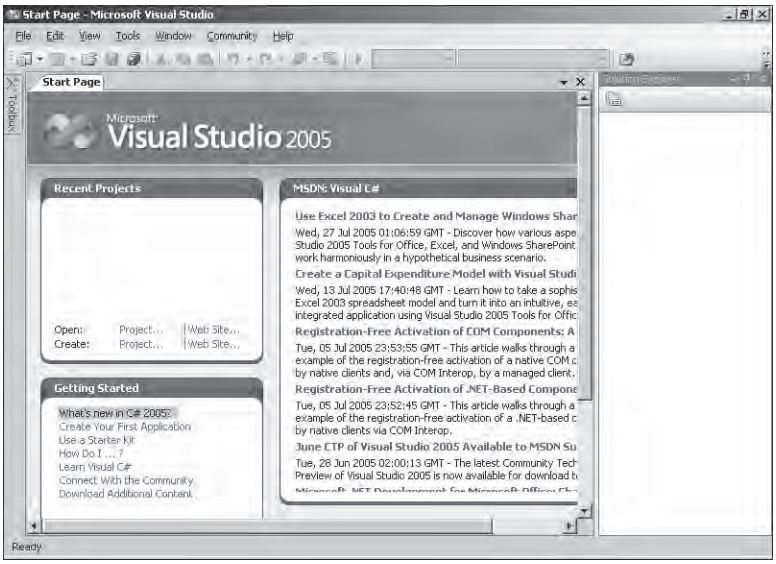
יצירת יישום console ב- Visual Studio 2005

1. בסביבת Microsoft Windows, לחץ Start והצבע על All Programs.

2. לחץ על הסמל של Microsoft Visual Studio 2005 כדי להפעיל את התוכנית.

הערה

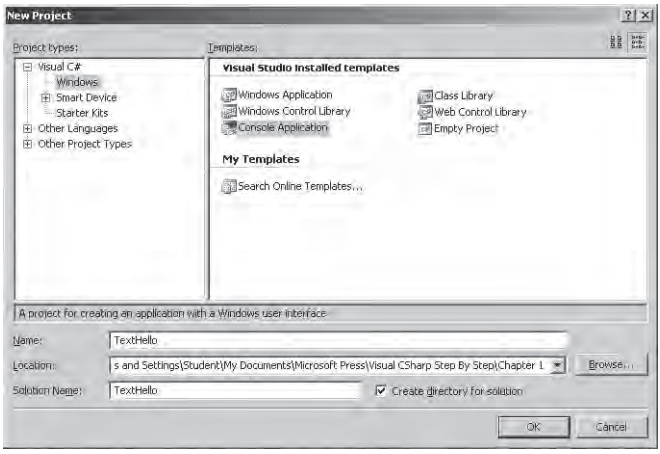
אם זו הפעם הראשונה שאתה מפעיל את Visual Studio 2005, ייתכן שתופיע תיבת דו-שיח שתבקש ממך לבחור את הגדרות ברירת המחדל עבור סביבת הפיתוח. Visual Studio 2005 יכולה להגדיר את עצמה על-פי שפת הפיתוח המועדפת עליך. תיבות הדו-שיח השונות שבסביבת הפיתוח המשולבת - IDE (Integrated Development Environment) תהיינה מקובעות על פי ברירת מחדל המותאמת לשפה שבחרת. מהרשימה בחר Visual C# Development Setting, ולאחר מכן לחץ Start Visual Studio. לאחר עיכוב קל החלון Visual Studio 2005 יוצג במסך.



3. בתפריט File, New בחר Project. תיפתח תיבת הדו-שיח New Project. תיבה זו מאפשרת ליצור פרויקט חדש המבוסס על מספר תבניות, כגון Windows Application, Class Library ו- Console Application, על פי סוג היישום שברצונך ליצור.

הערה

מגוון התבניות הזמינות תלויות בגרסת Visual Studio 2005 אשר ברשותך. קיימת אפשרות ליצור תבניות נוספות, אולם לא נרחיב בנושא זה.



4. בחלונית התבניות, לחץ על הסמל Console Application.

5. בשדה Location, הקלד: **C:\Documents and Settings***YourName***\My Documents\Microsoft Press\Visual CSharp Step by Step\Chapter 1**

החלף את *YourName* עם שם המשתמש שלך ב-Windows. כדי לחסוך בכתיבה נתייחס מעתה ואילך לנתיב "C:\Documents and Settings*YourName*\My Documents" בתור התיקייה "\My Documents".

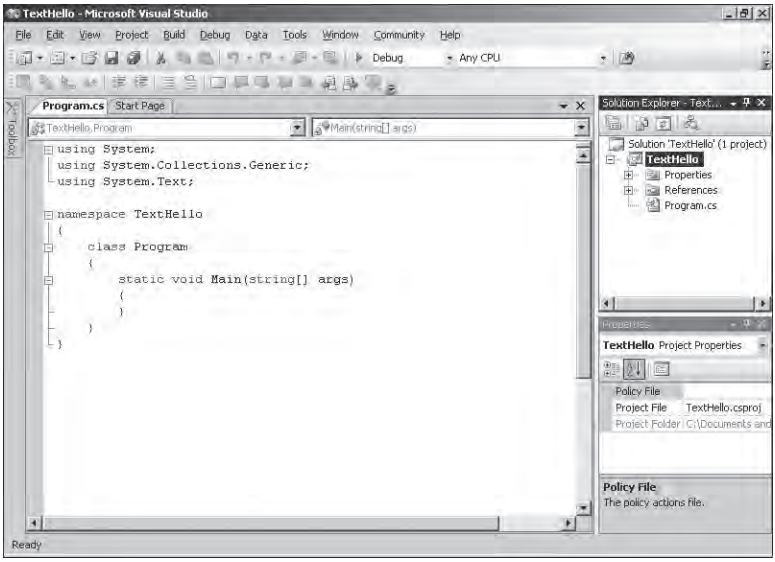
הערה

אם שם התיקייה שהזנת אינו קיים, Visual Studio 2005 תיצור אותה עבורך.

6. בשדה Name הקלד **TextHello**.

7. ודא שתיבת הסימון Create Directory for Solution אכן מסומנת, ולחץ OK. הפרויקט החדש יפתח.

שורת התפריטים שבראש המסך מאפשרת גישה לכלים השונים אשר ישמשו אותך בסביבת הפיתוח. ניתן לגשת לתפריטים ולשלוט בהם באמצעות העכבר או המקלדת, בדומה לכל תוכנה אחרת הפועלת תחת Windows. *סרגל הכלים* נמצא תחת שורת התפריטים ומכיל לחצנים המהווים קיצורי דרך לפקודות נפוצות. החלון **Code and Text Editor**, התופס חלק גדול מתצוגת ה-IDE, מציג את התוכן של קבצי המקור (source files) בפרויקטים מרובי קבצים, לכל קובץ מקור יש כרטיסייה (tab) משלו המסומנת בשם הקובץ. לחץ על הכרטיסייה פעם אחת כדי להציג את קובץ המקור בקדמת החלון Code and Text Editor. בחלון **Solution Explorer** מוצגים שמות הקבצים הקשורים לפרויקט, לצד מספר פריטים אחרים. לחיצה-כפולה על אחת משמות הקבצים בחלון זה תגרום להצגת תוכן קובץ המקור בחלון Code and Text Editor.



לפני כתיבת הקוד, בחן קודם את הקבצים המופעים בחלון Solution Explorer אשר נוצרו על ידי Visual Studio 2005 כחלק מהפרויקט שלך:

- **Solution 'TextHello'** זהו קובץ פתרון עליון (top-level solution file). לכל יישום יש קובץ אחד מסוג זה. אם תשתמש ב- Windows Explorer כדי לפתוח את התיקייה \My Documents\Microsoft Press\Visual CSharp Step by Step\Chapter 1\TextHello, תראה שהשם האמיתי של קובץ זה הוא TextHello.sln. כל קובץ פתרון כולל הפניה לקובץ פרויקט אחד לפחות.
- **TextHello** זהו קובץ הפרויקט של C#. כל קובץ פרויקט מפנה לפחות לאחד או יותר מהקבצים המכילים את קוד המקור ופריטים אחרים עבור הפרויקט. קוד המקור של פרויקט מסוים חייב להיות כתוב כולו בשפת תכנות אחידה. בסביבת Windows Explorer, קובץ זה מופיע בתור TextHello.csproj, ונמצא בתיקייה \My Documents\Microsoft Press\Visual CSharp Step by Step\Chapter 1\TextHello\TextHello.
- **Properties** זו תיקייה בפרויקט TextHello. אם תפתח אותה, תראה שהיא מכילה קובץ בשם AssemblyInfo.cs. קובץ זה הינו קובץ מיוחד המשמש להוספת תכונות (**attributes**) לתוכנית, כגון שם המחבר, תאריך כתיבת התוכנית וכו'. יש תכונות נוספות המשמשות לשינוי אופן הפעולה של התוכנית. ספר זה אינו עוסק בתכונות אלו.
- **References** זו תיקייה המכילה הפניות לקוד מהודר (compiled code) אשר יכול לשמש את היישום שלך.

- **Program.cs** זהו קובץ מקור של #C, אשר מוצג ראשון בחלון Code and Text Editor בעת יצירת הפרויקט. בקובץ זה תכתוב את הקוד. הוא מכיל קוד קצר אשר נכתב באופן אוטומטי על ידי Visual Studio 2005, אשר נבחן מייד.

התוכנית הראשונה שלך

הקובץ Program.cs מגדיר מחלקה בשם Program המכילה שיטה (class) בשם **Main**. כל השיטות חייבות להיות מוגדרות במסגרת מחלקה. השיטה **Main** הינה מיוחדת בכך שהיא מציינת את נקודת הכניסה (entry point) של התוכנית. היא חייבת להיות שיטה סטטית (static). על שיטות בכלל נלמד בפרק 3, על שיטות מסוג static נלמד בפרק 7, ועל השיטה Main נלמד בפרק 11.

חשוב

שפת #C מבחינה בין אותיות גדולות וקטנות (case-sensitive). על כן צריך להקפיד בכתיבת המילים. עליך לכתוב, למשל, **Main** ולא main (מתכנתים רבים משקיעים את זמנם באיתור תקלות הנובעות מכתיבה שאינה אחידה - שים לב).

בתרגיל הבא תכתוב קוד שיגרום להציג את ההודעה **Hello World** על המסך. תבנה ותפעיל את יישום המסך, ותלמד כיצד ניתן להשתמש במרחבי שמות (namespaces) כדי ליצור זהויות.

כתוב קוד באמצעות טכנולוגיית IntelliSense

1. בחלון Code and Text Editor, אשר מציג את הקובץ Program.cs, הצב את הסמן בין הפקודות של השיטה **Main**, מייד לאחר הסוגר המסולסל השמאלי, והקלד **Console**. לאחר הקלדת ה-C של המילה **Console** תופיע רשימה אשר מכילה את כל מילות המפתח (keywords) וסוגי הטיפוסים (data types) הרלוונטיים להקשר הנוכחי. באפשרותך להמשיך ולהקליד, או למצוא את הפריט Console וללחוץ עליו לחיצה-כפולה באמצעות העכבר. לחילופין, לאחר הקלדת **Con**, הרשימה תתביית באופן אוטומטי על הפריט **Console**, אם אין מילה אחרת המתחילה בשלוש אותיות אלו, ותוכל לבחור בו באמצעות הקשת Enter, Spacebar או Tab. טכנולוגיה זו נקראת IntelliSense (זו רגישות להקשת אותיות ומילים).

על השיטה **Main** להיראות כך:

```
static void Main(string[] args)
{
   Console
}
```

הערה

Console הינה מחלקה מובנית המכילה את השיטות אשר תפקידן להדפיס הודעות על המסך ולקבל קלט מהמקלדת.

2. הקלד נקודה (.) מייד אחרי **Console**. רשימת IntelliSense נוספת תציג את השיטות (methods), המאפיינים (properties) והשדות (fields) מהמחלקה Class.

3. דפדף ברשימה, בחר בשיטה **WriteLine** והקש Enter. לחילופין, באפשרותך להמשיך להקליד עד ש-**WriteLine** יבחר אוטומטית ואז להקיש Enter.

הרשימה תיסגר, והשיטה **WriteLine** תתווסף לקובץ המקור. כעת, השיטה **Main** תוצג כך:

```
static void Main(string[] args)
{
   Console.WriteLine
}
```

4. הקלד סוגר שמאלי, ורשימת IntelliSense תופיע על המסך.

רשימה זו מציג את הפרמטרים של השיטה **WriteLine**, שהינה למעשה שיטה מועמסת (**overloaded method**). משמעות הדבר היא שהמחלקה **Console** מכילה יותר משיטה אחת בשם **WriteLine**. כל גרסה שלה מיועדת להדפסת סוגים שונים של נתונים (על שיטות מועמסות ראה בפרק 3). השיטה **Main** תוצג כך:

```
static void Main(string[] args)
{
   Console.WriteLine(
}
```

באפשרותך ללחוץ על חצי הרשימה כדי לדפדף בין הגרסאות המועמסות של **WriteLine**.

5. הקלד סוגר ימני ולאחריו הנקודה-פסיק.

```
static void Main(string[] args)
{
   Console.WriteLine();
}
```

6. הקלד את המחרוזת "**Hello World**" בין הסוגריים.

השיטה **Main** תוצג כך:

```
static void Main(string[] args)
{
   Console.WriteLine("Hello World");
}
```

טיפ

כדאי להתרגל להקליד יחדיו תווים שבאים בזוגות, כמו זוגות של סוגריים רגילים או מסולסלים, ולא לחכות לסיום כתיבת התוכן שביניהם. לעיתים שוכחים את התו הסוגר, כאשר מסיימים את הקלדת התוכן.

סמלים של IntelliSense

טכנולוגיית IntelliSense מציגה את שמות כל חברי (members) המחלקה. לצד שם של כל חבר מופיע סמל המייצג את סוג החבר. להלן הסמלים השונים ופירושם:

סמל	עיקרי
	C# keyword
	method (discussed in Chapter 3)
	property (discussed in Chapter 14)
	class (discussed in Chapter 7)
	struct (discussed in Chapter 9)
	enum (discussed in Chapter 9)
	interface (discussed in Chapter 12)
	delegate (discussed in Chapter 16)
	Namespace

הערה

לעיתים קרובות תבחין בשורות קוד הכוללות שני לוכסנים שלאחריהם טקסט רגיל. אלו הן הערות. המהדר (compiler) אינו מתייחס אליהן, אולם הן שימושיות ביותר כי בצורה זו ניתן לתעד מה התוכנית צריכה לבצע. לדוגמה:

```
Console.ReadKey(); // Wait fot the user to press a key
```

המהדר יתעלם מכל הטקסט שאחרי שני הלוכסנים ועד סוף השורה. באפשרותך גם להוסיף הערות ארוכות משורה אחת, אך עליך להתחיל את השורה ב- */. המהדר יתעלם מהטקסט עד שיתקל ברצף /*, אשר ניתן לכתוב אותו לאחר מספר בלתי מוגבל של שורות. רצוי להוסיף למסמך כמה שיותר הערות.

בנה והפעל יישום console

1. בתפריט Build לחץ Build Solutions. פעולה זו תגרום להידור של הקוד #C, וליצירת תוכנית הניתנת להפעלה (executable). החלון Output יופיע מתחת לחלון Code and Text Editor.

טיפ

אם החלון Output אינו מופיע, פתח את תפריט View ולחץ Output.

בחלון Output, הודעות דומות לזו שלהלן מציגות כיצד התוכנית עוברת הידור ואת פרטי השגיאות (אם ישנן). במקרה זה לא אמורות להיות שגיאות או אזהרות, והתוכנית אמורה להיבנות בצורה מוצלחת.

```
------ Build started: Project: TextHello, Configuration: Debug Any CPU ------
 Csc.exe /noconfig /nowarn:1701,1702 /errorreport:prompt /warn:4 …

 Compile complete -- 0 errors, 0 warnings

 TextHello -> C:\Documents and Settings\krudo\My Documents\Microsoft Press…
========== Build: 1 succeeded or up-to-date, 0 failed, 0 skipped ==========
```

הערה

כוכבית לאחר שם הקובץ בכרטיסיה שמעל החלון Code and Text Editor מעידה על כך שהיה שינוי בקובץ מאז השמירה האחרונה. אין צורך לשמור את הקובץ לפני הבנייה, כי הפקודה Build Solution עושה זאת באופן אוטומטי.

2. בתפריט Debug בחר Start Without Debugging. חלון Command ייפתח והתוכנית תופעל. בחלון תוצג ההודעה **Hello World**, והתוכנית תושהה עד שהמשתמש ילחץ על מקש כלשהו, כפי שניתן לראות להלן:

3. ודא שהחלון Command שהתוכנית מוצגת בו נמצא בקדמת המסך, ולאחר מכן הקש Enter. חלון Command ייסגר ואתה תחזור לסביבת הפיתוח של Visual Studio 2005.

הערה

כאשר תפעיל את התוכנית באמצעות הפקודה Start Debugging שבתפריט Debug, היישום יפעל, אולם החלון Command ייסגר ללא השהיה, לפני שתלחץ על מקש כלשהו.

4. ב- Solution Explorer, לחץ על הפרויקט TextHello (ולא על הפתרון), ולאחר מכן לחץ Show All Files. ענפים נוספים בעץ הפרויקט נוספו בשם bin ו-obj. רשומות הן התיקיות bin ו-obj אשר בתיקיית הפרויקט (My Documents\Microsoft Press\Visual CSharp Step\Chapter 1\TextHello\TextHello). תיקיות אלו נבנו בעת יצירת היישום, והן מכילות את הגרסה הניתנת להפעלה (executable) של התוכנית ומספר קבצים נוספים.

5. ב- Solution Explorer, לחץ על הפלוס (+) שמשמאל לרשומה bin. כעת תופיע תיקייה נוספת בשם Debug.

6. ב- Solution Explorer, לחץ על הפלוס (+) שמשמאל לרשומה Debug. כעת תופענה שלוש תיקיות: TextHello.exe, TextHello.pdb ו-TextHello.vshost.exe. הקובץ TextHello.exe הינו התוכנית המהודרת, וזהו למעשה הקובץ שיופעל כשתבחר Start Without Debugging בתפריט Debug. שני הקבצים האחרים מכילים מידע המשמש את Visual Studio 2005 כשתפעיל את התוכנית במצב Debug (כשתבחר Start Debugging בתפריט Debug).

הידור באמצעות שורת פקודה

באפשרותך להדר את קבצי המקור כדי ליצור קבצי הפעלה באופן ידני בעזרת שורת הפקודה csc. תחילה עליך לבצע את הפעולות הבאות, כדי להכין את סביבת הפיתוח:

1. ב-Windows היכנס לתפריט Start ← Microsoft Visual Studio 2005 ← Visual Studio Tools ולחץ על Visual Studio 2005 Command Prompt. ייפתח חלון Command, ומשתני הסביבה **PATH**, **LIB** ו-**INCLUDE** יוגדרו באופן שיכללו את מיקום הספריות והעזרים השונים של .NET Framework.

טיפ

ניתן גם להפעיל את התסריט (script) vcvarsall.bat, שבתיקייה C:\Program Files\Microsoft Visual Studio 8\VC folder, אם ברצונך להגדיר את משתני הסביבה דרך חלון Command Prompt רגיל.

2. בחלון Visual Studio 2005 Command Prompt, הקלד את הפקודה הבאה בתיקייה \My Documents\Microsoft Press\Visual CSharp Step by Step\Chapter 1\TextHello\TextHello project folder:

```
cd C:\Documents and Settings\meir\My Documents\Microsoft Press\
Visual CSharp Step By Step\Chapter 1\TextHello\Text Hello
```

3. הקלד את הפקודה הבאה:

```
csc /out:TextHello.exe Program.cs
```

פקודה זו יוצרת את קובץ ההפעלה TextHello.exe מקובץ המקור של C#. אם לא תשתמש בשורת-הפקודה /out, קובץ ההפעלה ישאב את שמו מקובץ המקור, וייקרא Program.exe.

4. הפעל את התוכנית על ידי הקלדת הפקודה

```
TextHello
```

התוכנית אמורה לעבוד באופן זהה קודמת, מלבד העובדה שההודעה "Press any key to continue" לא תופיע הפעם.

שימוש במרחבי שמות

הדוגמה שהוצגה הייתה תוכנית קטנה מאוד. תוכניות קטנות עשויות להתרחב לתוכניות גדולות יותר, ואז עשויות להיות שתי בעיות. ראשית, כמות גדולה של קוד (פקודות) גורמת בדרך כלל קשיי הבנה ותפעול, בהשוואה לתוכנית קטנה; ושנית, יותר קוד משמעו גם יותר שמות עבור נתונים, שיטות ומחלקות. ככל שמספר השמות גדל, כך עולה הסבירות להתנגשות של שמות ולתקלות שונות (במיוחד כאשר התוכנית מפעילה ספריות של ספקי תוכנה אחרים, צד-שלישי).

מתכנתים נהגו בעבר לפתור את בעיית התנגשות השמות על ידי הוספת סיווג כלשהו או קידומת (או מערך של סיווגים) לפני השם שקבעו. הבעיה היא שפתרון זה מסורבל וגורם לשמות ארוכים מאוד. בסופו של דבר, המתכנת נאלץ להקדיש זמן רב להקלדה וקריאה של שמות ארוכים ובלתי נהירים, ביחס לזמן המוקדש לכתיבת התוכנה עצמה.

מרחבי שמות (Namespaces) תורמים לפתרון בעיה זו באמצעות יצירת מיכל (Container) עבור מזהים (identifiers), כגון מחלקות (classes). שתי מחלקות בעלות אותו שם לא תתנגשנה זו בזו אם הן מאוחסנות בשני מרחבי שמות בעלי שמות שונים. ניתן ליצור מחלקה בשם **Greeting** בתוך מרחב השמות **TextHello** בדרך זו:

```
namespace TextHello
{
    class Greeting
    {
        ...
    }
}
```

כעת תוכל להתייחס למחלקה Greeting בתור **TextHello.Greeting** במסגרת התוכניות שתבנה. אם מישהו אחר יוצר גם הוא מחלקה **Greeting** במרחב שמות שונה ומתקין אותה במחשב שלך, התוכניות שלך ימשיכו לפעול כהלכה כי הן משתמשות במחלקה **TextHello.Greeting**. אם ברצונך להשתמש במחלקה **Greeting** החדשה, עליך לציין שאתה מתכוון למחלקה הנמצאת במרחב השמות החדש. כלומר, צריך לכתוב את שם מרחב השמות לפני שם המחלקה, נקודה, ואחר כך לכתוב את שם המחלקה.

ניתן ליצור מרחב שמות מקונן, שבו במרחב שמות ישנו מרחב שמות נוסף, שבו נמצאת המחלקה.

כדאי להגדיר את כל המחלקות במסגרות של מרחבי שמות, וכך גם פועלת סביבת Visual Studio 2005 המשתמשת בשם הפרויקט שלך כמרחב השמות העליון (top-level). ערכת פיתוח התוכנה SDK (Software Development Kit) של NET Framework., פועלת גם היא בצורה זו. כל מחלקה בתוך NET Framework. שוכנת בתוך מרחב שמות. לדוגמה, המחלקה **Console** שוכנת במרחב השמות **System**. משמעות הדבר היא שהשם המלא של המחלקה הוא **System.Console**.

כמובן, שאם היה עליך לרשום כל פעם את שם המחלקה המלא, אזי היה אפשר פשוט לקרוא למחלקה **SystemConsole** ולוותר על השימוש במרחבי שמות. המזל, שניתן לפתור בעיה זו באמצעות ההנחיה (directive) **using**. אם תחזור לתוכנית TextHello ב- Visual Studio 2005, ובחלון Code and Text Editor תסתכל בקובץ Program.cs, תבחין במשפט הבא:

```
using System;
using System.Collections.Generic;
using System.Text;
```

מטרת ההנחיה using היא להכריז על השימוש במרחב שמות כלשהו, כדי שאחך כך לא יהיה עוד צורך לכתוב את שם מרחב השמות הרלוונטי לצד האובייקטים השונים. שלושת מרחבי השמות שלעיל מכילים מחלקות אשר השימוש בהן שכיח ביותר, ולכן Visual Studio 2005 מוסיפה את ההנחיות **using** הללו באופן אוטומטי בכל פעם שנוצר פרויקט חדש. באפשרותך להוסיף הנחיות **using** נוספות בתחילת קובץ המקור.

התרגיל הבא מדגים ומרחיב את הסבר על מרחבי השמות.

התנסה בשמות שאינם מקוצרים

1. בחלון Code and Text Editor, הפוך את ההנחיה using אשר בראש הקובץ Program.cs להערה:

```
// using System;
```

2. בתפריט Build לחץ Build Solution. הבנייה נכשלת, ובחלונית Output תופיע הודעת שגיאה פעמיים, פעם אחת עבור כל שימוש במחלקה **Console**:

```
The name 'Console' does not exist in the current context.
```

3. בחלונית Output, לחץ לחיצה-כפולה על הודעת השגיאה, והמזהה (identifier) אשר גרם לשגיאה יסומן בקובץ המקור Program.cs.

טיפ

השגיאה הראשונה עשויה להשפיע על אמינות הודעות האבחון שיוצגו אחריה. אם הבנייה גרמה ליותר מהודעת אבחון אחת, תקן תחילה רק את הראשונה מביניהן, התעלם מהאחרות, ונסה לבנות שוב. אסטרטגיה זו פועלת בצורה הטובה ביותר אם קבצי המקור שלך יהיו קטנים ותפעל באופן אחיד, תוך כדי בנייה תכופה.

4. בחלון Code and Text Editor, ערוך את השיטה Main באופן שתשתמש בשם המלא **System.Console**. השיטה **Main** תוצג כך:

```
System.Console.Write("Hello World");
```

הערה

שים לב שכאשר אתה מקליד **.System** שמות כל הפריטים במרחב השמות **System** מוצגים ברשימה IntelliSense.

5. בתפריט Build לחץ על Build Solution. הפעם, הבנייה מוצלחת. אם לא, ודא שהמחלקה **Main** *זהה במדויק* לתצורה שלה בקוד הקודם, ונסה לבנות שוב.

6. הפעל את היישום כדי לוודא שהוא עדיין פועל, בלחיצה על Start Without Debugging בתפריט Debug.

ב- Solution Explorer, לחץ על הפלוס (+) שמשמאל לרשומה References. הפעולה תגרום להצגת האסופות (assemblies) שה- Solution Explorer מפנה אליהן. אסופה הינה תיקייה המכילה קוד שנכתב על ידי מפתחים אחרים (כגון .NET Framework). במקרים מסוימים, המחלקות במרחב שמות מאוחסנות באסופה בעלת שם זהה לזה של מרחב השמות (כגון **System**), אבל לא בהכרח - יש אסופות אשר מכילות יותר ממרחב שמות אחד. בכל פעם שאתה משתמש במרחב שמות, עליך לוודא שהפנית לאסופה המכילה את המחלקות השייכות למרחב השמות הנכון. אחרת, התוכנית שלך לא תיבנה (או תופעל).

יש שימוש נוסף למילת המפתח using, שפירושו מתן כינוי שייכות (alias). הקוד הבא מדגים מהו כינוי. במקום להשתמש ב- System.Console.WriteLine, משתמשים בכינוי myConsole.

```
using System.Collections.Generic;
using System.Text;
using myConsole = System.Console;
namespace UsingAlias
{
    class Program
    {
        static void Main(string[] args)
        {
            myConsole.WriteLine("Hello Console World.");
        }
    }
}
```

יצירת יישום Windows Forms

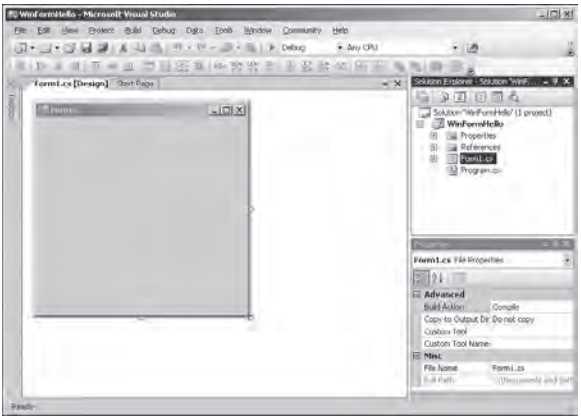
עד כה השתמשת ב- Visual Studio 2005 כדי ליצור ולהפעיל יישומיי console בסיסיים. סביבת הפיתוח של Visual Studio 2005 כוללת גם את כל האמצעים הדרושים ליצירת

יישומיי Windows גרפיים. באפשרותך לעצב את ממשק המשתמש של יישומיי Windows באופן אינטראקטיבי באמצעות Visual Designer. לאחר מכן, Visual Studio 2005 תיצור את משפטי התוכנית עבור ישום ממשק המשתמש אשר עיצבת. מדברים אלה ניתן להבין ש- Visual Studio 2005 מאפשרת לצפות בשתי תצוגות של היישום: תצוגת העיצוב (Design View) ותצוגת הקוד (Code View). החלון Code and Text Editor (המציג את משפטי התוכנית) משמש גם בתור תצוגת העיצוב (המאפשרת לתכנן את ממשק המשתמש), ובאפשרותך לעבור בין שני המצבים בכל זמן נתון.

בתרגילים הבאים תלמד ליצור תוכנית Windows בחסות Visual Studio 2005. תוכנית זו תציג טופס (form) פשוט שמכיל תיבת טקסט שתוכל להקליד בה את שמך, ולחצן אשר לחיצה עליו תגרום להופעת ברכת שלום אישית בתיבת ההודעות. עליך להשתמש בתצוגת העיצוב כדי ליצור את ממשק המשתמש על ידי מיקום פקדים (controls) בטופס; בדיקת הקוד שיצרה Visual Studio 2005; שימוש בתצוגת העיצוב לשינוי תכונות הבקרה; שימוש בתצוגת העיצוב לשינוי גודל הטופס; כתיבת הקוד שיצור את התגובה ללחיצה על הלחצן; והפעלת תוכנית Windows הראשונה שלך.

פרויקט Windows ב- Visual Studio 2005

1. היכנס לתפריט File ← New ולחץ Project. תיבת הדו-שיח New Project תיפתח.
2. בחלונית Project Types לחץ Visual C#.
3. בחלונית Templates לחץ על הסמל Windows Application.
4. ודא שהשדה Location מפנה לתיקייה **\My Documents\Microsoft Press\ Visual CSharp Step by Step\Chapter 1**.
5. בשדה Name הקלד **WinFormHello**.
6. ודא שהשדה Solution נמצא במצב **Create new Solution**. פעולה זו גורמת ליצירת פתרון חדש לאחסון יישום Windows. האלטרנטיבה, **Add to Solution**, תגרום לצירופו של הפרויקט לפתרון TextHello.
7. לחץ OK. סביבת הפיתוח Visual Studio 2005 תסגור את היישום הנוכחי (לפני כן ייתכן שתתבקש לשמור אותו), וגם תיצור ותציג טופס Windows ריק בחלון תצוגת העיצוב.



בתרגיל הבא עליך להשתמש בתצוגת העיצוב כדי להוסיף שלושה פקדים לטופס Windows. לאחר מכן תבחן חלק מהקוד אשר נוצר על ידי C# באופן אוטומטי באמצעות Visual Studio 2005, כדי ליישם את הפקדים שהוספת.

צור את ממשק המשתמש

1. בחר בכרטיסייה Toolbox שמשמאל לטופס בתצוגת העיצוב. ערכת הכלים תופיע מעל הטופס, ועליה הרכיבים והפקדים השונים שניתן למקם בטופס Windows, אם אינך מוצא את הכרטיסייה, היכנס לתפריט View ולחץ על Toolbox (באפשרותך להשתמש גם בצירוף המקשים Ctrl + Alt + X).
2. בערכת הכלים, לחץ על הפלוס (+) שליד Common Controls כדי להציג את רשימת הפקדים המשמשים את מרבית יישומיי Windows Forms.
3. לחץ Label, ולאחר מכן לחץ על החלק הגלוי של הטופס. פקד **Label** יתווסף לטופס וערכת הכלים תיעלם מהתצוגה.

טיפ

אם ברצונך להשאיר את ערכת הכלים מבלי להסתיר את הטופס, לחץ על הלחצן Auto Hide שפינה הימנית-עליונה של ערכת הכלים (נראה כמו סיכה). ערכת הכלים תוצג באופן קבוע בצד השמאלי של חלון Visual Studio 2005, ותצוגת העיצוב תצטמצם בהתאם (אם וברשותך מסך ברזולוציה נמוכה, פעולה זו עשויה לעלות לך בשטח מסך רב). לחיצה נוספת על Auto Hide תביא להעלמת ערכת הכלים מהמסך.

4. סביר להניח שהפקד **Label** אינו נמצא בדיוק במקום שרצית אותו. באפשרותך לשנות את מיקום הפקדים שבטופס על ידי גרירתם. גרור את הפקד **Label** לאזור הפינה השמאלית-עליונה של הטופס, למרות שביישום זה המיקום המדויק אינו משמעותי.
5. בתפריט View, לחץ Properties Window. החלון Properties יופיע בצד הימני של המסך. חלון זה מאפשר להגדיר את מאפייני הפריטים שבפרויקט. הוא מתאים עצמו להקשר, כלומר הוא יודע להציג את המאפיינים של הפריט הנבחר. אם תלחץ על הטופס עצמו בתצוגת העיצוב, תראה שהחלון Properties יציג את מאפייני הטופס עצמו. אם תלחץ על הפקד **Label**, החלון יעבור להצגת המאפיינים של פקד זה.
6. לחץ על הפקד **Label** שבטופס. בחלון Properties שנה את המאפיין Text מ-**label1** ל- **Enter your name** והקש Enter. הטקסט של התווית שבטופס ישתנה כעת ל- **Enter your name**.

טיפ

על-פי ברירת המחדל, המאפיינים מסווגים לקטגוריות. אם אתה מעדיף להציגם בסדר אלפביתי, לחץ על הלחצן Alphabetical שמעל רשימת המאפיינים.

7. הצג שוב את ערכת הכלים. לחץ על **TextBox**, ואחר כך על הטופס. פקד **TextBox** יתווסף לטופס. גרור אותו כדי להציבו בדיוק תחת הפקד **Label**.

טיפ

בעת גרירת פקד על הטופס, סימונים ישרים יופיעו בכל פעם שהפקד נמצא בקו ישר, אופקי או אנכי עם אחד הפקדים האחרים. סימונים אלה מאפשרים למקם את הפקדים בצורה ישרה ואסתטית.

8. סמן את הפקד **TextBox**, ובמאפיין **Text** שבחלון Properties הקלד **here** והקש Enter. המילה **here** תופיע בתיבת הטקסט שבטופס.

9. אתר את המאפיין (**Name**) בחלון Properties. סביבת הפיתוח Visual Studio 2005 מקצה שמות ברירת מחדל לפקדים ולטפסים, אשר ממלאים את תפקידם, אולם חסרי משמעות והקשר. שנה את שם הפקד **TextBox** ל-**userName**.

הערה

בפרק 2 נדון בהרחבה במוסכמות הקיימות להקצאת שמות לפקדים ומשתנים.

10. הצג שוב את ערכת הכלים. לחץ על **Button**, ולאחר מכן על הטופס. מקם את הפקד Button מימין לפקד **TextBox** כדי ששני הפקדים יהיו מיושרים אופקית זה לזה.

11. באמצעות החלון Properties, שנה את המאפיין **Text** של הפקד **Button** ל-**OK**. את המאפיין (**Name**) שנה ל-**ok**. הכיתוב על הלחצן שבטופס ישתנה כעת.

12. בחלון תצוגת העיצוב, לחץ על הטופס Form1. שים לב לבקרי הגודל (ריבועים קטנים) בצלע התחתונה, בצלע הימנית ובפינה הימנית תחתונה של הטופס.

13. מקם את סמן העכבר על הבקר הפינתי. הסמן יהפוך לחץ אלכסוני דו-צידי.

14. החזק את לחצן העכבר השמאלי וגרור את הסמן כדי לשנות את גודל הטופס. הפסק לגרור ושחרר את לחצן העכבר כאשר המרווח סביב הבקרים פחות או יותר שווה.

טיפ

ניתן לשנות את גודל רוב הפקדים לאחר מיקומם על הטופס באמצעות סימון הפקד וגרירה של אחד מבקרי הגודל המופיעים בפינותיו. שים לב שגודלם של חלק מהפקדים, כגון Label, נקבע אוטומטית על-פי התוכן שלהם, והוא אינו ניתן לשינוי באמצעות גרירה.

על הטופס להיות כעת דומה לתרשים שלהלן:

15. ב- Solution Explorer, לחץ לחיצה ימנית על הקובץ Form1.cs ואחר כך לחץ View Code. קובץ המקור Form1.cs יופיע כעת בחלון Code and Text Editor.

כעת יש שתי כרטיסיות בשם Form1.cs מעל החלון Code and Text Editor/Design View. בכל עת ניתן ללחוץ על הקובץ שלאחריו כתוב [Design], כדי לחזור לחלון תצוגת העיצוב.

הכרטיסייה Form1.cs מכילה חלק מהקוד שנוצר באופן אוטומטי על ידי Visual Studio 2005. שים לב למרכיבים הבאים:

- **הנחיות using** סביבת הפיתוח Visual Studio 2005 הוסיפה מספר הנחיות **using** בראש קובץ המקור (יותר הנחיות מאשר בדוגמה הקודמת). למשל:

```
using System.Windows.Forms;
```

מרחבי השמות (namespaces) הנוספים מכילים את המחלקות והפקדים המשמשים לבניית יישום גרפי, כגון המחלקות **TextBox**, **Label** ו-**Button**.

- **מרחב השמות** השם אשר Visual Studio 2005 הקצתה למרחב השמות העליון (top-level) הוא שם הפרויקט:

```
namespace WinFormHello
{
    ...
}
```

- **מחלקה** Visual Studio 2005 הוסיפה מחלקה בשם **Form1** בתוך מרחב השמות **WinForm Hello**:

```
namespace WinFormHello
{
    public partial class Form1...
    {
        ...
    }
}
```

הערה

בשלב זה, התעלם ממילת המפתח partial שבמחלקה זו. נבין אותה בהמשך.

מחלקה זו מבצעת את הטופס אשר ייצרת בתצוגת העיצוב (בפרק 7 נלמד על מחלקות).

הקוד היחיד שנראה כאן הוא הבנאי (**Constructor**), בנאי הינו שיטה מיוחדת בעלת שם זהה לזה של המחלקה. הוא מופעל בעת יצירת הטופס ועשוי להכיל קוד שצריך לאתחל (**initialize**) את הטופס (פרק 7 עוסק בבנאים). אולם, Visual Studio 2005 מסתירה מאתנו כמה דברים, כפי שנראה כעת.

ביישום Windows Forms, סביבת הפיתוח Visual Studio 2005 יוצרת למעשה כמות קוד גדולה. הקוד מבצע פעולות כגון יצירה והצגה של הטופס בעת הפעלת היישום, וגם יוצר ומציב את הפקדים השונים בטופס. אולם, קוד זה עשוי להשתנות בכל פעם שאתה מוסיף בקרים לטופס או משנה את מאפייניהם. אינך צריך לשנות את הקוד (אי-לכך כנראה שכל שינוי שתבצע ישירות בקוד, יימחק בפעם הבאה שתערוך את הטופס בתצוגת העיצוב), ולכן Visual Studio 2005 מסתירה אותו מפניך.

כדי להציג את הקוד המוסתר, חזור ל- Solution Explorer, ולחץ Show All Files. שים לב שכעת מופיע פלוס (+) ליד Form1.cs. כשתלחץ על הפלוס, תראה שני קבצים: Form1.Designer.cs ו- Form1.resx.

לחיצה-כפולה על הקובץ Form1.Designer.cs תגרום להצגת התוכן שלו בחלון Code and Text Editor. בקובץ זה תמצא את שאר הקוד שנכתב עבור המחלקה Form1. שפת #C מאפשרת לפצל את הקוד של מחלקה כלשהי למספר קבצי מקור, כל עוד כל חלק מהמחלקה מסומן באמצעות מילת המפתח partial תוך שמירה על שם מחלקה זהה. קובץ זה כולל אזור המכונה **Windows Form Designer generated code**. לחיצה על הפלוס (+) שליד האזור תביא להצגת הקוד אשר Visual Studio 2005 יוצר ומתחזק בכל פעם שאתה עורך טופס בעזרת חלון תצוגת העיצוב. התוכן הממשי של הקובץ כולל:

- **השיטה InitializeComponent** שיטה זו מוזכרת בקובץ Form1.cs. הקוד בשיטה זו מגדיר את מאפייני הפקדים שהוספת לקובץ בתצוגת העיצוב (על שיטות נלמד בפרק 3). להלן מספר משפטים מהשיטה התואמים לפעולות שביצעת באמצעות החלון Properties:

```
...
private void InitializeComponent()
{
   this.label1 = new System.Windows.Forms.Label();
   this.userName = new System.Windows.Forms.TextBox();
   this.ok = new System.Windows.Forms.Button();
   ...
   this.label1.Text = “Enter Your Name”;
   ...
   this.userName.Text = “here”;
```

```
   ...
   this.ok.Text = “OK”;
   ...
}
...
```

- **שלושה שדות** Visual Studio 2005 יצרה שלושה שדות בתוך המחלקה **Form1**. שדות אלה מופיעים לקראת סוף הקובץ:

```
private System.Windows.Forms.Label label1;
private System.Windows.Forms.TextBox userName;
private System.Windows.Forms.Button ok;
```

השדות גורמים לביצוע שלושת הפקדים אשר הוספת לטופס בתצוגת העיצוב (על שדות נלמד בפרק 7).

נציין שוב שלמרות שמעניין להביט בתוכן הקובץ, לעולם אל תשנה את התוכן בעצמך. Visual Studio 2005 יוצרת את הקובץ מחדש בכל פעם שאתה עורך שינויים בתצוגת העיצוב. קוד שעליך לכתוב מקומו בקובץ Form1.cs.

בשלב זה ייתכן ואתה תוהה היכן נמצאת השיטה **Main** וכיצד מוצג הטופס בעת הפעלת היישום. זכור שהשיטה **Main** מגדירה את הנקודה שבה תתחיל התוכנית. ב- Solution Explorer, יש קובץ מקור נוסף בשם Program.cs. לחיצה-כפולה על הקובץ תביא להופעת הקוד הבא בחלון Code and Text Editor:

```
namespace WinFormHello
{
   static class Program
   {
      /// <summary>
      /// The main entry point for the application.
      /// </summary>
      [STAThread]
      static void Main()
      {
         Application.EnableVisualStyles();
         Application.Run(new Form1());
      }
   }
}
```

ניתן להתעלם ממרבית הקוד, מלבד המשפט המרכזי:

```
Application.Run(new Form1());
```

משפט זה יוצר את הטופס ומציג אותו כאשר היישום מופעל.

בתרגיל הבא תלמד להוסיף קוד אשר מופעל כשהמשתמש לוחץ על הלחצן OK שבטופס.

כתוב קוד עבור הלחצן OK

1. לחץ על הכרטיסייה Form1.cs[Design] שמעל החלון Text and Code Editor, כדי להציג את Form1 בתצוגת העיצוב.
2. הצב את סמן העכבר על הלחצן OK אשר בטופס, ולחץ עליו לחיצה-כפולה. קובץ המקור Form1.cs יופיע בחלון Code and Text Editor. כתוצאה, Visual Studio 2005 תוסיף את השיטה **ok_Click** למחלקה Form1. גם יתווסף משפט לשיטה **InitializeComponent** בקובץ Form1.Designer.cs, שמטרתו לקרוא ל-**ok_Click** כאשר המשתמש לוחץ OK. הדבר נעשה על ידי **delegate**. על נציגים (**delegates**) נלמד בפרק 16.
3. הקלד את המשפט **MessageBox** המוצג להלן בתוך השיטה **ok_Click**. הקוד של השיטה נראה כעת כך:

```
private void ok_Click(object sender, EventArgs e)
{
   MessageBox.Show("Hello " + userName.Text);
}
```

ודא שהקלדת את הקוד בדיוק כפי שמוצג לעיל, כולל הנקודה-פסיק בסוף המשפט.

כעת אתה מוכן להפעיל את התוכנית Windows הראשונה שלך.

הפעל את התוכנית Windows

1. בתפריט Debug לחץ Start Without Debugging. Visual Studio 2005 תשמור את השינויים שביצעת, תהדר את התוכנית ותפעיל אותה. הטופס Windows יוצג כעת.
2. הקלד את שמך ולחץ OK. על המסך תופיע תיבת הודעה אשר תברך אותך לשלום.

3. לחץ OK בתיבת ההודעה. התיבה תיסגר.
4. סגור את החלון Form1 (בלחיצה על לחצן ה-X בפינה הימנית-עליונה של הטופס).

☯ אם ברצונך להמשיך לפרק הבא

השאר את סביבת הפיתוח Visual Studio 2005 פעילה, ועבור לפרק 2.

☯ אם ברצונך לסגור את Visual Studio 2005

בתפריט File בחר Exit. אם מופיעה תיבת דו-שיח Save, לחץ Yes כדי לשמור את עבודתך.

פרק 1 – טבלה מסכמת

המשימה	צריך	קיצור
ליצור יישום console חדש	בתפריט File ← New לחץ Project כדי לפתוח את תיבת הדו-שיח New Project. בסוג הפרויקט, בחר Visual C# ובתבנית בחר Console Application. קבע תיקייה עבור קבצי הפרויקט בתיבה Location. קבע בשם לפרויקט, ולחץ OK.	
ליצור יישום Windows חדש	בתפריט File ← New לחץ Project כדי לפתוח את תיבת הדו-שיח New Project. בסוג הפרויקט בחר Visual C# ובתבנית בחר Windows Application. קבע תיקייה עבור קבצי הפרויקט בתיבה Location. קבע בשם לפרויקט ולחץ OK.	
להדר יישום	בתפריט Build לחץ Build Solution.	F6
להפעיל יישום	בתפריט Debug לחץ Start Without Debugging	Ctrl+F5

פרק 2

עבודה עם משתנים, אופרטורים וביטויים

בסיום פרק זה, תוכל:

- להבין מהם משפטים (statements), מזהים (identifiers) ומילות מפתח (keywords).
- להשתמש במשתנים (variables) לשמירת נתונים ומידע.
- לעבוד עם סוגי נתונים פרימיטיביים (primitive data types).
- להשתמש באופרטורים אריתמטיים, כגון פלוס (+) ומינוס (-).
- להגדיל ולהקטין משתנים.

בפרק 1 למדת להשתמש בסביבת הפיתוח של Microsoft Visual Studio 2005 כדי לבנות ולהפעיל יישומיי מסוף (console) ו- Windows Forms. בפרק זה תלמד על מרכיבי התחביר (syntax) והסמנטיקה של Microsoft Visual C#, כולל משפטים (statements), מילות מפתח (keywords) ומזהים (identifiers). תכיר את הסוגים הפרימיטיביים המובנים בשפת C# ואת מאפייני הערכים של כל סוג. גם תלמד כיצד להכריז על משתנים מקומיים ולהשתמש בהם (משתנים המתקיימים רק במסגרת של פונקציה או מקטע קוד קצר), תלמד על האופרטורים האריתמטיים שמספקת C#, תלמד להשתמש באופרטורים כדי לשנות ערכים ולשלוט בביטויים המכילים שני אופרטורים או יותר.

הבנת מבני משפטים

משפט (statement) הוא פקודה המבצעת פעולה. משפטים נמצאים בתוך שיטות. תלמד עוד על שיטות בפרק 3, אולם כעת חשוב על שיטה בתור קטע קוד בעל שם מזהה המכיל רצף משפטים ונמצא בתוך מחלקה. השיטה **Main**, אשר הוצגה בפרק הקודם, היא דוגמה טובה לכך. משפטים ב-C# חייבים לקיים מספר חוקים מוגדרים היטב, אשר נקראים **תחביר** (**syntax**). כלל הפעולות שמשפטים אלה מבצעים נקרא **סמנטיקה** (semantics). אחד מחוקי התחביר הפשוטים והחשובים ביותר של C# קובע שעליך לסיים כל משפט בנקודה-פסיק (;). משפט שאין בסופו נקודה-פסיק לא יעבור את שלב ההידור (compilation):

```
Console.WriteLine("Hello World");
```

טיפ

#C היא שפת תכנות ב"תבנית חופשית". משמעות הדבר היא שלרווחים על ידי מקשי Space ו-Tab, או שבירת שורות על ידי Enter אין משמעות, מעבר להיותם מפרידים. במילים אחרות, ניתן לעצב את המשפטים בכל סגנון רצוי, כדי שהתוכנית תהיה ברורה וקלה לקריאה והבנה. עם זאת, רצוי לשמור על סגנון פשוט ועקבי.

כדי לתכנת ברמה טובה, בכל שפת תכנות שהיא, יש ללמוד את התחביר והסמנטיקה שלה ולהשתמש בשפה בצורה טבעית ואחידה. גישה זו תניב תוכנית קלה לקריאה ונוחה לשינוי. בספר זה תמצא דוגמאות לכתיבה נכונה של המשפטים החשובים בשפת #C.

מזהים ושימושיהם

מזהים (**identifiers**) הם השמות המשמשים לזיהוי האלמנטים השונים בתוכניות שלך. בשפת #C עליך לקחת בחשבון את חוקי התחביר הבאים בעת בחירת מזהים:

- ניתן להשתמש באותיות לועזיות (גדולות או קטנות), ספרות וקו-תחתי (_) בלבד.
- מזהה חייב להתחיל באות (קו-תחתי נחשב לאות).

לדוגמה, **result**, **_score**, **footballTeam** ו-**plan9** הם מזהים תקינים, בעוד ש-**result%**, **footballTeam$** ו-**9plan** אינם תקינים.

חשוב

שפת #C רגישה לגודל האותיות ומבחינה בין אותיות רישיות ("גדולות") לבין אותיות רגילות ("קטנות"). לפיכך, המזהים **TotalNumbers** ו-**totalNumbers** **אינם** זהים זה לזה.

מילות מפתח

שפת #C משריינת 77 מזהים לשימוש השפה, ולא ניתן להשתמש בהם למטרות אחרות. אלה הם **מילות מפתח** (**keywords**), ולכל אחת מהן יש משמעות ייחודית. דוגמאות למילות מפתח הן **class**, **namespace** ו-**using**. בספר זה תלמד את משמעות רוב מילות המפתח. הטבלה הבאה מפרטת את כל מילות המפתח בסדר אלפביתי.

abstract	as	base	bool
break	byte	case	catch
char	checked	class	const
continue	decimal	default	delegate
do	double	else	enum
event	explicit	extern	false
finally	fixed	float	for
foreach	goto	if	implicit
in	int	interface	internal

is	lock	long	namespace
new	null	object	operator
out	override	params	private
protected	public	readonly	ref
return	sbyte	sealed	short
sizeof	stackalloc	static	string
struct	switch	this	throw
true	try	typeof	uint
ulong	unchecked	unsafe	ushort
using	virtual	void	volatile
while			

טיפ

בחלון Code and Text Editor של Visual Studio 2005, כאשר תקליד מילת מפתח, היא תוצג **בכחול**.

משתנים

משתנה (**variable**) הוא מקום אחסון עבור ערך. ניתן לראות את המשתנה כתיבה המכילה נתון בסיסי או תוצאה, לשמירה זמנית במהלך פעולת התוכנית. עליך להקצות לכל משתנה בתוכנית שם ייחודי. שם המשתנה משמש לשם פנייה אל הערך שהוא מכיל. למשל, אם ברצונך לאחסן את ערך העלות של מוצר מסוים בחנות, תוכל ליצור משתנה בשם **productCost** כדי לשמור בו את עלות המוצר. תוכל לקרוא את תוכן המשתנה productCost, שהינו למעשה מחיר המוצר אשר הזנת קודם.

הקצאת שמות למשתנים

רצוי לאמץ את המוסכמות עבור הקצאת שמות למשתנים, כדי להימנע מחוסר בהירות ואי-הבנות. להלן מספר המלצות כלליות בנושא זה:

- אל תשתמש בקו-תחתי (_).
- אל תיצור משתנים אשר הדבר היחיד המבדיל ביניהם הוא אותיות רישיות ורגילות. כלומר, אל תיצור משתנה בשם **myVariable** ומשתנה נוסף בשם **MyVariable** באותה תוכנית. הדבר עשוי לבלבל. צריך להיות עקבי בקביעת השמות.

הערה

שימוש במשתנים המובדלים זה מזה על ידי אותיות רישיות בלבד מגביל את היכולת להשתמש במחלקות ביישומים אשר פותחו באמצעות שפות אחרות אשר אינן רגישות לרישיות (case sensitive), כגון Visual Basic.

- התחל את המשתנה באות קטנה.
- במזהים המורכבים ממספר מילים, רצוי שהמילים הבאות תתחלנה באות רישית לשם הבלטה (שיטה זו נקראת סימון **camelCase**, כמו לדוגמה: bankAccountNumber).
- אל תשתמש בשיטת סימון הונגרית (למשתמשי Visual C++ היא מוכרת. אם אינך מכיר אותה, שכח מזה!).

חשוב

רצוי להתייחס לשתי ההמלצות הראשונות כמחייבות, כי הן מתייחסות ליכולת של התוכנית לעמוד במפרט הדרישות לשימוש בשפות תכנות - CLS (Common Language Specification). אם ברצונך לכתוב תוכניות שיכולות לפעול גם עם שפות אחרות, כגון Microsoft Visual Basic .NET, עליך להתחשב בהמלצות אלו.

לדוגמה שמות המשתנים **score**, **footballTeam**, **_score** ו-**FootballTeam** הם חוקיים ב-C#, אולם רק שני הראשונים מקיימים את המלצות CLS.

הכרזה על משתנים

משתנים (variables) דומים לתיבות זיכרון שמסוגלות להכיל ערך. ל-C# יש סוגים רבים של ערכים הניתנים לאחסון ולעיבוד כגון שלמים (integers), מספרי נקודה-צפה (floating-point numbers) ומחרוזות תווים (strings). בעת הכרזה על משתנה עליך לציין את סוג הנתונים שהוא יאחסן.

הערה

מתכנתים מרקע תכנות של Microsoft Visual Basic צריכים לשים לב לכך ש-C# אינה מאפשרת הכרזות מרומזות או משתמעות. עליך להכריז באופן חד-משמעי על כל המשתנים לפני השימוש בהם, כדי שהקוד יוכל לעבור הידור.

שם וסוג המשתנה נקבעים במשפט ההכרזה (declaration statement). לדוגמה, המשפט שלהלן מכריז שהמשתנה **age** מכיל ערכים מסוג **int** (שלם-integer). בסוף המשפט יש לכתוב נקודה-פסיק (;).

```
int age;
```

סוג המשתנה **int** הוא אחד מסוגי המשתנים **הפרימיטיביים**, **integer**, והוא מייצג מספר שלם. (בהמשך הפרק תלמד על סוגים פרימיטיביים נוספים). לאחר הכרזת המשתנה, ניתן להציב בו ערך. המשפט הבא מציב במשתנה **age** את הערך 42. גם כאן דרוש התו נקודה-פסיק.

```
age = 42;
```

הסימן שווה (=) הוא אופרטור הצבה (**assignment**), שתפקידו להעביר את הערך שמימינו אל המשתנה שמשמאלו. לאחר ההצבה, ניתן להשתמש במשתנה **age** במסגרת הקוד, ולמעשה להתייחס לערך המאוחסן בו. המשפט הבא מציג את ערך המשתנה **age**, 42, על המסך.

```
Console.WriteLine(age);
```

טיפ

אם תשאיר את סמן העכבר על משתנה כלשהו בחלון Code and Text Editor, תופיע מסגרת קטנה עם סוג המשתנה.

סוגי נתונים פרימיטיביים

ב-#C יש מספר של סוגים מובנים הנקראים **סוגי נתונים פרימיטיביים** (**Primitive Data Types**). הטבלה הבאה מציגה את סוגי הנתונים הפרימיטיביים השכיחים ביותר ב-#C, ואת טווח הערכים שניתן להציב בהם.

Data type	Description	Size (bits)	*Range	Sample usage
int	Whole numbers	32	<–>2^{31} through 2^{31}<–>1	int count; count = 42;
long	Whole numbers (bigger range)	64	<–>2^{63} through 2^{63}<–>1	long wait; wait = 42L;
float	Floating-point numbers	32	$\pm 3.4 \times 10^{38}$	float away; away = 0.42F;
double	Double precision (more accurate) floating-point numbers	64	$\pm 1.7 \times 10^{308}$	double trouble; trouble = 0.42;
decimal	Monetary values	128	28 significant figures	decimal coin; coin = 0.42M;
string	Sequence of characters	16 bits per character	Not applicable	string vest; vest = "42";
char	Single character	16	0 through 2^{16} <–>1	char grill; grill = '4';
bool	Boolean	8	true or false	bool teeth; teeth = false;

* The value of 2^{16} is 65,536; the value of 2^{31} is 2,147,483,648; and the value of 2^{63} is 9,223,372,036,854,775,808.

משתנים מקומיים ללא ערך

לאחר הכרזה על משתנה, הוא יכיל ערך אקראי עד אשר תציב בו ערך. תכונה זו של משתנים הייתה מקור לבאגים רבים בתוכניות C ו-++C, אשר יצרו משתנה והשתמשו בו לפני שהוצב בו ערך כלשהו. שפת #C אינה מאפשרת זאת. עליך להציב ערך במשתנה לפני השימוש בו, אחרת תתקבל שגיאת הידור. דרישה זו נקראת **Definite Assignment Rule**.

לדוגמה, המשפטים הבאים יגרמו לשגיאה בזמן ההידור, כי לא הוצב ערך במשתנה **age**:

```
int age;
Console.WriteLine(age); // compile time error
```

הצגת ערכים מסוג 'נתון פרימיטיבי'

התרגיל הבא מדגים כיצד פועלים מספר סוגי נתונים פרימיטיביים באמצעות תוכנית #C הנקראת PrimitiveDataTypes.

הצג ערכים מסוג נתון פרימיטיבי

1. הפעל את Visual Studio 2005.
2. בתפריט File ← Open בחר Project/Solution.

   תיבת הדו-שיח Open Project תוצג במסך.
3. פתח את התיקייה \My Documents\Microsoft Press\Visual CSharp Step by Step\Chapter 2\Primitive-DataTypes. סמן את הקובץ PrimitiveDataTypes.sln ולחץ Open.

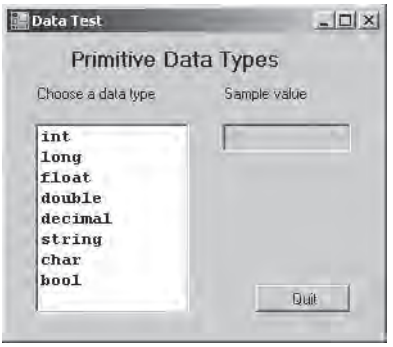
   כעת הפתרון (solution) ייטען, ואילו Solution Explorer תציג את הפתרון ואת הפרויקט PrimitiveDataTypes.

הערה

שמות קבצי פתרון נושאים את הסיומת sln., כמו למשל PrimitiveDataTypes.sln. פתרון עשוי להכיל פרויקט אחד או יותר. קבצי פרויקט נושאים את הסיומת csproj. אם תפתח פרויקט במקום פתרון, סביבת הפיתוח Visual Studio 2005 תיצור עבורו באופן אוטומטי קובץ פתרון חדש. אם תבנה את הפתרון, יישמרו באופן אוטומטי עדכונים או קבצים חדשים אם יש כאלה, ותתבקש לספק שם ומקום אחסון עבור קובץ הפתרון החדש.

4. בתפריט Debug לחץ Start Without Debugging.

   חלון היישום הבא יוצג במסך:



5. בחר את הסוג **string** מתיבת הרשימה Choose A Data type.

   בתיבה Sample value יופיע הערך 42.
6. בחר מהרשימה את הסוג **int**.

בתיבה Sample value יופיע כעת הערך **to do**, שמשמעותו שיש לכתוב את המשפט אשר יציג ערך **int**.

7. בחר כל אחד מסוגי הנתונים שברשימה. גם המשפט עבור הסוגים **double** ו-**bool** טרם נכתב.

8. לחץ Quit כדי לסגור את החלון ולעצור את התוכנית.
השליטה חוזרת לסביבת הפיתוח של Visual Studio 2005.

השתמש בסוגי נתונים פרימיטיביים בקוד עצמו

1. לחץ לחיצה-ימנית על הקובץ Form1.cs אשר ב- Solution Explorer ואז לחץ View Code.
החלון Code and Text Editor ייפתח ויציג את הקובץ Form1.cs.

2. בחלון Code and Text Editor, אתר את השיטה showFloatValue שלהלן:

```
private void showFloatValue()
{
   float var;
   var = 0.42F;
   value.Text = “0.42F”;
}
```

טיפ

כדי לאתר פריט בפרויקט, היכנס לתפריט Edit ← Find and Replace ובחר Quick Find. בתיבת הדו-שיח שתופיע תישאל מה ברצונך לחפש. הקלד את שם הפריט ברצונך למצוא ולחץ Find Next. לפי ברירת המחדל, החיפוש אינו רגיש רישיות (case sensitive). על כן, אם ברצונך לבצע חיפוש רגיש-רישיות, לחץ על הפלוס (+) שליד התווית Find Options כדי להציג אפשרויות נוספות, וסמן את תיבת הסימון Match Case. נסה גם את האפשרויות האחרות שבתיבת הדו-שיח.

מקש הקיצור לפתיחת תיבת הדו-שיח Quick Find הוא Ctrl+F. ניתן להשתמש בקיצור Ctrl+H לפתיחת תיבת הדו-שיח Quick Find and Replace.

השיטה **showFloatValue** מופעלת בעת בחירת הסוג **float** מתיבת הרשימה. שיטה זו מכילה שלושה משפטים:

- המשפט הראשון מכריז על משתנה float בשם var.
- המשפט השני מציב את הערך 0.42F. ה-F הוא תוספת סוג, המציינת שיש לראות ב-0.42 ערך מסוג float. אם תשמיט את ה-F, הערך 0.42 ייחשב ערך מסוג double, והידור התוכנית לא יבוצע, כי בדרך זו לא ניתן להציב ערך מסוג מסוים במשתנה מסוג אחר.
- המשפט השלישי מציג את ערך המשתנה בתיבת הטקסט **value** שבטופס. יש מספר נקודות המחייבות התייחסות במשפט זה. אופן ההצגה של הפריט בתיבת הטקסט נקבע על ידי המאפיין **Text** של התיבה (ביצעת זאת כבר בפרק 1 כאשר עיצבת טופס באמצעות החלון Properties). משפט זה מדגים כיצד לעשות זאת בתכנות, על ידי הביטוי **value.Text**. הנתון שתזין למאפיין **Text** חייב להיות מחרוזת (רצף של תווים), ולא

מספר. אם תנסה להציב מספר במאפיין **Text**, התוכנית לא תהודר. מסיבה זו, המשפט מציג את הטקסט "0.42F" בתיבת הטקסט (כל מה שמופיע בין גרשיים כפולים נחשב טקסט, או **מחרוזת**). ביישום אמיתי, תצטרך להוסיף משפט אשר ממיר את ערך המשתנה var למחרוזת, ואחר כך מציב אותו במאפיין **Text**. לשם כך עליך לדעת יותר על C# ו- NET Framework. (פרקים 11 ו-19 עוסקים בין השאר בהמרות של סוגי נתונים).

.3 בחלון Code and Text Editor, אתר את השיטה **showIntValue** שלהלן:

```
private void showIntValue()
{
   value.Text = "to do";
}
```

השיטה **showIntValue** מופעלת בעת בחירת הסוג **int** מתיבת הרשימה.

טיפ

יש דרך נוספת למצוא שיטה בחלון Code and Text Editor. לחץ על הרשימה Members שמעל החלון מצד ימין. בחלון תוצג רשימה של כל השיטות (ופריטים אחרים). לחיצה על שם של אחד הפריטים תעביר אותך אליו בחלון Code and Text Editor.

.4 הקלד את שני המשפטים הבאים בתחילת השיטה **showIntValue**, בין הסוגריים המסולסלים:

```
int var;
var = 42;
```

השיטה **showIntValue** תוצג כך:

```
private void showIntValue()
{
   int var;
   var = 42;
   value.Text = "to do";
}
```

.5 בתפריט Build בחר Build Solution.

הבנייה תגרום למספר אזהרות, אך לא לשגיאות. לעת עתה התעלם מן האזהרות.

.6 במשפט המקורי, שנה את המחרוזת **"to do"** ל-**"42"**.

השיטה תוצג כעת בדיוק כך:

```
private void showIntValue()
{
   int var;
   var = 42;
   value.Text = "42";
}
```

7. בתפריט Debug בחר Start Without Debugging.

הטופס (form) יופיע שוב.

טיפ

אם ערכת את קובץ המקור מאז הבנייה האחרונה של הפרויקט, הפקודה Start Without Debugging תבנה מחדש את התוכנית באופן אוטומטי לפני הפעלת היישום.

8. בחר את הסוג **int** מתיבת הרשימה. ודא שבתיבת הטקסט Sample value מופיע הערך 42.

9. לחץ Quit כדי לסגור את החלון ולעצור את התוכנית.

10. בחלון Code and Text Editor, אתר את השיטה **showDoubleValue**.

11. ערוך את השיטה **showDoubleValue** בדיוק כך:

```
private void showDoubleValue()
{
   double var;
   var = 0.42;
   value.Text = “0.42”;
}
```

12. בחלון Code and Text Editor, אתר את השיטה **showBoolValue**.

13. ערוך את השיטה **showBoolValue** בדיוק כך:

```
private void showBoolValue()
{
   bool var;
   var = false;
   value.Text = “false”;
}
```

14. בתפריט Debug לחץ Start Without Debugging.

הטופס יופיע שוב.

15. בתיבת הרשימה בחר את הסוגים **int**, **double** ו-**bool**. בכל אחד מהמקרים ודא שהערך הנכון אכן מופיע בתיבת הטקסט Sample value.

16. לחץ Quit כדי לעצור את התוכנית.

אופרטורים אריתמטיים

שפת #C מאפשרת פעולות חשבוניות רגילות: סימן הפלוס (+) לחיבור, סימן מינוס (-) לחיסור, כוכבית (*) לכפל ולוכסן (/) לחילוק. סמלים אלה הם **אופרטורים** (operators) מכיוון שהם פועלים (באנגלית: operate) על ערכים כדי ליצור ערך חדש. בדוגמה הבאה, המשתנה moneyPaidToConsultant יכיל בסופו של דבר את המכפלה של 750 (המחיר ליום עבודה) ב-20 (מספר הימים שהיועץ הועסק):

```
long moneyPaidToConsultant;
moneyPaidToConsultant = 750 * 20;
```

הערה

הערכים שהאופרטור פועל עליהם הם **אופרנדים** (operands). בביטוי 20 * 750, הכוכבית היא האופרטור ומערכים 750 ו-20 הם האופרנדים.

קביעת ערך האופרטור

לא ניתן ליישם את כל סוגי האופרטורים על כל סוגי הנתונים. האפשרות להפעיל אופרטור על ערך כלשהו תלויה בסוג הערך. לדוגמה, ניתן להשתמש בכל סוגי האופרטורים האריתמטיים על ערכים מסוג **char**, **int**, **long**, **float**, **double** או **decimal** המייצגים מספרים. אולם, מלבד מקרה יוצא-דופן אחד, אינך יכול להשתמש באופרטורים אריתמטיים על ערכים מסוג **string** או **bool** (שהינם טקסט). על כן, המשפט הבא אינו חוקי, כי הסוג **string** אינו תומך באופרטור מינוס (אין כל משמעות בהפחתה אריתמטית של מחרוזת אחת מאחרת):

```
// compile time error
Console.WriteLine("Tel Aviv" - "Holon");
```

יוצא הדופן הוא שניתן להשתמש באופרטור פלוס (+) כדי לשרשר מחרוזות. המשפט הבא יציג במסך 431, ולא 44, מכיוון שגם "43" וגם "1" הן מחרוזות:

```
Console.WriteLine("43" + "1");
```

טיפ

ניתן להשתמש בשיטה **int32.Parse** כדי להמיר ערכים מסוג string ל-integer אם צריך לבצע חישובים אריתמטיים של ערכים המאוחסנים כמחרוזות.

חשוב לדעת שסוג הערך המופק מהחישובים האריתמטיים תלוי בסוג האופרנדים ששימשו לחישוב. לדוגמה, ערך הביטוי 2.0 / 5.0 הוא 2.5. שני האופרנדים הם מסוג **double** (ב-#C, מספרים ליטרליים עם נקודה עשרונית לעולם יהיו מסוג **double**, ולא **float**, כדי לשמור על רמת דיוק גבוהה ככל הניתן). לפיכך, גם סוג התוצאה יהיה **double**, אולם ערך הביטוי 2 / 5 הוא 2 כי אלה מספרים מסוג **int**, כי שפת #C תמיד מעגלת ערכים כלפי מטה.

העניין מסתבך כאשר משלבים אופרנדים מסוגים שונים. לדוגמה, הביטוי 2.0 / 5 מורכב מ-**int** ומ-**double**. המהדר של #C מזהה את חוסר ההתאמה ויוצר קוד שתפקידו להמיר את הערך **int** ל-**double** לפני ביצוע הפעולה. תוצאת האופרטור תהיה לפיכך מסוג **double** (2.5).

אולם למרות שהדבר אפשרי, רצוי לקבוע מראש מה שרוצים להשיג ולא לערבב אופרנדים מסוגים שונים סתם כך.

שפת #C תומכת גם באופרטור אריתמטי מוכר פחות: אופרטור השארית (remainder/modulus), המיוצג על ידי סימן האחוז (%). התוצאה של x % y תהיה השארית הנותרת לאחר חלוקת x ב-y. לדוגמה, 2 % 9 שווה ל-1 כי 9 חלקי 2 שווה ל-4 עם שארית 1.

הערה

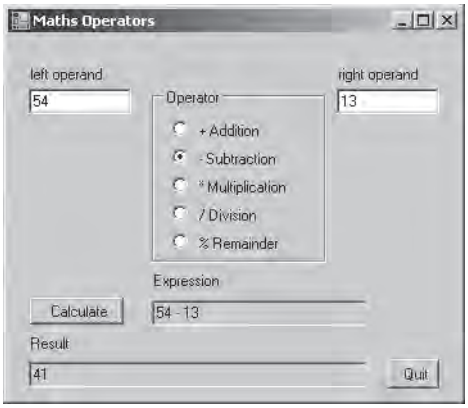
ב-C וב-++C לא ניתן להשתמש באופרטור % על ערכים מסוג floating-point, אולם ב-#C ניתן לעשות זאת.

בחינת אופרטורים אריתמטיים

בתרגיל הבא מוצגת התוכנית **MathOperators** כדי להדגים כיצד ניתן להשתמש באופרטורים אריתמטיים על ערכים מסוג **int**.

שימוש באופרטורים אריתמטיים

1. בתפריט File ← Open בחר Project/Solution. פתח את הפרויקט **MathsOperators** שבתיקייה \My Documents\Microsoft Press\Visual CSharp Step by Step\Chapter 2\ MathsOperators.
2. בתפריט Debug בחר Start Without Debugging.
   כתוצאה יוצג במסך טופס (Form).
3. בתיבת הטקסט עבור האופרנד השמאלי הקלד **54**.
4. בתיבת הטקסט עבור האופרנד הימני הקלד **13**.
   כעת ניתן ליישם כל אחד מהאופרטורים על האופרנדים שבתיבות הטקסט.
5. לחץ על האפשרות 'Subtraction -' ולחץ Calculate.
   הטקסט בתיבה Expression ישתנה ל- '54 – 13', ובתיבה Result יופיע הערך 41, כפי שניתן לראות להלן:



6. לחץ על האפשרות 'Division /' , ולחץ Calculate.

הטקסט בתיבה Expression ישתנה ל- '13 / 54', ובתיבה Result יופיע הערך 4. החישוב 45 חלקי 31 מניב את התוצאה 4.153846 (בקירוב), אבל בשפת #C, כאשר מחלקים שלם (integer) בשלם מתקבל שלם, ולכן התוצאה היא 4.

7. בחר באפשרות 'Remainder %' , ולחץ Calculate.

הטקסט בתיבה Expression ישתנה ל- '13 % 54', ובתיבה Result יופיע הערך 2. הסיבה לכך היא שהשארית המתקבלת מהחלוקה של 54 ב-13 היא 2 (הביטוי (13 * (13 / 54)) – 54 שווה ל-2 כאשר נעגל כלפי מטה לערך שלם בכל אחד מהשלבים).

8. נסה שילובים נוספים של מספרים ואופרטורים. לסיום לחץ Quit.

התוכנית תיפסק, ותחזיר אותך לסביבת הפיתוח של Visual Studio 2005.

כעת נתבונן בקוד התוכנית MathsOperators.

בחן את קוד התוכנית MathsOperators

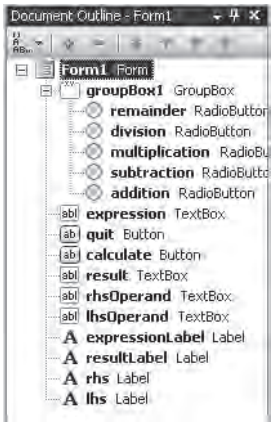
1. הצג את הטופס Form1 בחלון תצוגת העיצוב (Design View) (ייתכן שתצטרך ללחוץ על הכרטיסייה [Form1.cs[Design).

טיפ

ניתן לעבור במהירות בין החלון Design View והחלון Code and Text Editor, אשר מציג את הקוד עבור הטופס. לשם כך הקש **F7**.

2. בתפריט View ← Other Windows בחר Document Outline (Ctrl + Alt+ T).

חלון Document Outline יופיע ויציג את השמות והסוגים של הפקדים הנמצאים בטופס. לחיצה על כל אחד מהפקדים שבטופס תגרום להארת שמו בחלון Document Outline.



חשוב

היזהר לא לשנות או למחוק בטעות שמות של פקדים שנמצאים בטופס בעת הצפייה בהם בחלון Document Outline. אם תעשה זאת, היישום יפסיק לפעול.

3. לחץ על שני הפקדים מסוג TextBox המשמשים להקלדת מספרים בטופס. בחלון Document Outline ודא שהם נקראים lhsOperand ו-rhsOperand.

כאשר הטופס מופעל, המאפיין **Text** של כל אחד משני הפקדים מכיל (כמחרוזת) את הערכים המספריים שהזנת.

4. ודא שהפקד **TextBox** המשמש להצגת הביטוי המחושב שבחלק התחתון של הטופס, נקרא **expression**, ושהפקד **TextBox** המשמש להצגת תוצאת החישוב נקרא **result**.

בזמן ההפעלה, הזנת ערך מחרוזת למאפיין **Text** של פקד מסוג **TextBox** תגרום להצגת הערך.

5. סגור את החלון Document Outline.

6. הקש F7 כדי להציג את קובץ המקור Form1.cs בחלון Code and Text Editor.

7. בחלון Code and Text Editor, אתר את השיטה subtractValues:

```
private void subtractValues()
{
   int lhs = int.Parse(lhsOperand.Text);
   int rhs = int.Parse(rhsOperand.Text);
   int outcome;
   outcome = lhs - rhs;
   expression.Text = lhsOperand.Text + " - " + rhsOperand.Text;
   result.Text = outcome.ToString();
 }
```

המשפט הראשון בשיטה זו מכריז על משתנה מסוג int בשם **lhs** ומאתחל אותו לתוצאת ההמרה של המאפיין **lhsOperand.Text** לערך int (מאפיין **Text** של **TextBox** הינו מחרוזת, ויש להמירו לשלם (integer) כדי שיהיה ניתן לאחסנו במשתנה מסוג **int**. זה מה שעושה השיטה **int.Parse**). המשפט השני מכריז על משתנה מסוג **int** בשם **rhs** ומאתחלת אותו לתוצאת ההמרה של המאפיין **rhsOperand.Text** לערך **int**. המשפט השלישי מכריז על משתנה מסוג **int** בשם **outcome**. המשפט הרביעי מפחית את ערך המשתנה **rhs** מערך המשתנה **lhs**, ומציב את התוצאה במשתנה **outcome**. המשפט החמישי משרשר שלוש מחרוזות (באמצעות האופרטור +) ומציב את התוצאה במאפיין **expression.Text**. המשפט השישי ממיר את הערך int שבמשתנה **outcome** למחרוזת באמצעות השיטה **ToString**, ומציב את המחרוזת במאפיין **result.Text**.

המאפיין Text והשיטה ToString

פקדים מסוג **TextBox** שנמצאים בטופס כוללים מאפיין **Text** המאפשר גישה לתכנים המוצגים. לדוגמה, הביטוי **result.Text** מתייחס לתוכן של תיבת הטקסט **result**. לתיבות טקסט יש מאפיינים רבים נוספים, כגון גודל תיבת הטקסט ומיקומה בטופס. בפרק 14 נדון בהרחבה בנושא המאפיינים.

בכל מחלקה יש שיטה **ToString** שמטרתה להמיר אובייקטים למחרוזות המייצגות אותם. בדוגמה הקודמת, השיטה **ToString** של האובייקט השלם **outcome**, שימשה להמרת הערך השלם של **outcome** לערך המחרוזת המקביל לו. המרה זו הכרחית כי הערך מוצג במאפיין **Text** של השדה **result** יכול להכיל מחרוזות בלבד. כאשר תיצור מחלקות משלך, תגדיר בעצמך כיצד לכתוב את השיטה **ToString**. על **שיטות** (methods) תלמד בפרק 3.

שליטה בקדימות

קדימות (**precedence**) מכתיבה את סדר החישוב של האופרטורים בביטוי (expression). הביטוי שלהלן משתמש באופרטורים חיבור (+) וכפל (*):

```
2 + 3 * 4
```

ביטוי זה עשוי להיות דו-משמעי. האם 3 מתחבר לאופרטור + שמשמאלו, או לאופרטור * שמימינו? סדר הפעולות משמעותי, כי הוא משפיע על התוצאה:

- כאשר האופרטור + יקבל קדימות על פני האופרטור *, הערך 3 יתקשר לאופרטור +, ותוצאת החיבור (3 + 2) תהווה את האופרנד השמאלי של האופרטור *. תוצאת הביטוי כולו תהיה 4 * 5, כלומר 20.
- כאשר האופרטור * יקבל קדימות על פני האופרטור +, הערך 3 יתקשר לאופרטור *, ותוצאת המכפלה (4 * 3) תהווה את האופרנד הימני של האופרטור +. תוצאת הביטוי כולו תהיה 12 + 2, כלומר 14.

בשפת #C, לאופרטורים כפל (*), חילוק (/) ושארית (%) יש קדימות על פני האופרטורים חיבור (+) וחיסור (-). הפתרון של הביטוי 4 * 3 + 2 הוא לפיכך 14. כאשר נציג אופרטורים חדשים בהמשך הלימוד, נסביר את מצב הקדימות שלהם.

ניתן להשתמש בסוגריים כדי לעקוף את הקדימות ולכפות על אופרנדים לחבור לאופרטורים בסדר שונה. לדוגמה, בביטוי שלהלן, הסוגריים מאלצים את המספרים 2 ו-3 לחבור לאופרטור +, ותוצאת החיבור (5) מהווה את האופרנד השמאלי של האופרטור * אשר יפיק את הערך 20.

```
(2 + 3) * 4
```

הערה

הביטוי **"סוגריים"** מתייחס ל-( ); הביטוי **"סוגריים מסולסלים"** מתייחס ל-{ }; והביטוי **"סוגריים מרובעים"** מתייחס ל-[ ].

שיוך לצורך חישוב ביטויים

קדימות של אופרטורים פותרת רק חלק מהבעיות שיש בחישובים. מה קורה כאשר ביטוי מכיל אופרטורים שונים בעלי קדימות זהה? במקרים כאלה יש חשיבות לשיוך. **שיוך** (associativity) הוא הכיוון (שמאל או ימין) שלפיו נקשרים האופרטורים שבביטוי. הביטוי הבא משתמש באופרטורים כפל וחילוק:

4 / 2 * 6

ביטוי זה עשוי להיות דו-משמעי. האם 2 מתחבר לאופרטור / שמשמאלו או לאופרטור * שמימינו? קדימותם של שני האופרטורים זהה, אולם יש משמעות לסדר הפעולות, כי יש שתי תוצאות אפשריות:

- אם 2 יתקשר לאופרטור /, תוצאת החלוקה (2 / 4) תהווה את האופרנד השמאלי של האופרטור *, ותוצאת הביטוי כולו תהיה 6 * (2 / 4), כלומר 12.
- אם 2 יתקשר לאופרטור *, תוצאת המכפלה (6 * 2) תהווה את האופרנד הימני של האופרטור /, ותוצאת הביטוי כולו תהיה (6 * 2) / 4, כלומר 4/12.

הקדימות של האופרטורים כפל וחילוק זהה ולכן הקדימות לכשעצמה אינה יכולה לקבוע האם 2 יתחבר לאופרטור * או לאופרטור /. אולם, לכל אופרטור יש גם שיוך אשר קובע באיזה אופן יש לחשבו. האופרטורים * ו-/ הינם בעלי שיוך שמאלי (left-associative). כלומר, הם מחושבים משמאל לימין. לכן, 2 / 4 יחושב לפני הכפל ב-6, ולכן התוצאה תהיה 12. כאשר נציג אופרטורים חדשים בהמשך, נציין גם את השיוך שלהם.

הגדלה והקטנה של משתנים

אם ברצונך להוסיף 1 לערך המשתנה, ניתן לעשות זאת על ידי האופרטור +:

```
count = count + 1;
```

אולם מתכנת מנוסה לא יכתוב קוד שכזה, אלא ישתמש בשיטת קיצור. הגדלת ערך משתנה ב-1 שכיחה למדי בשפות תכנות, ולכן גם ב-C# ניתן לעשות זאת באמצעות הקיצור ++. להגדלת המשתנה count ב-1, כתוב את המשפט הבא:

```
count++;
```

גם הקטנת ערך משתנה ב-1 שכיחה מאוד וניתן לעשות זאת באמצעות הקיצור - - (מינוס כפול). להקטנת המשתנה **count** ב-1, כתוב את המשפט הבא:

```
count--;
```

הערה

האופרטורים '++' ו- '- -' הינם מסוג **unary**. כלומר, יכולים לקבל אופרנד אחד בלבד, ולכן הם בעלי קדימות ושיוך שמאלי זהה לזה של האופרטור ! (גם הוא מסוג unary) שנדון בו בפרק 4.

הטבלה הבאה מסבירה כיצד להשתמש בשני האופרטורים הללו.

Don't write this	Write this
variable = variable + 1;	variable++;
variable = variable - 1;	variable--;

Postfix ו-Prefix

אופרטור ההגדלה (++) ואופרטור ההקטנה (- -) אינם שגרתיים מבחינה זו שניתן לכתוב אותם לפני או אחרי המשתנה. במצב prefix של האופרטור, סמל האופרטור מוצב לפני המשתנה, ובמצב postfix סמל האופרטור מוצב אחרי המשתנה. להלן מספר דוגמאות:

```
 count++;  // postfix increment
 ++count;  // prefix increment
 count--;  // postfix increment
--count;   // prefix increment
```

מיקומו של האופרטור אינו משפיע על המשתנה המוגדל או המוקטן. לדוגמה, אם תכתוב **count++**, ערך המשתנה count יגדל ב-1, ואם תרשום **++count** ערך המשתנה count יגדל ב-1 גם הפעם. אם כך, מדוע אנו זקוקים לשני מצבים המבצעים את אותה פעולה? כדי לענות על שאלה זו יש לזכור שאופרטורים אלה, וכל האופרטורים האחרים מחזירים ערך. הערך המוחזר על ידי **count++** הוא ערך המשתנה count לפני הגדלתו, בעוד הערך המוחזר על ידי **++count** הוא ערך המשתנה count לאחר הגדלתו. להלן דוגמה:

```
int x;
x = 42;
Console.WriteLine(x++); // x is now 43, 42 written out
x = 42;
Console.WriteLine(++x); // x is now 43, 43 written out
```

שורת הקוד השלישית מציגה את x (שערכו 42) ולאחר מכן מוסיפה לו 1. כלומר, הערך שיוצג הוא 42 למרות שערכו כבר גדל. שורת הקוד החמישית מוסיפה 1 ל-x ורק אחר כך היא מציגה את ערכו, ולכן נקבל 43.

דרך טובה לזכור מה עושה כל מצב היא להסתכל על סדר האלמנטים (האופרנד והאופרטור) בביטוי מסוג prefix או postfix. בביטוי ++**x**, המשתנה **x** קודם לאופרטור, ולכן מוצג ערכו לפני שהוא מוגדל. בביטוי **x**++, האופרטור קודם למשתנה ולכן ערכו גדל ורק אחר כך מוצג.

אופרטורים אלה משמשים לעיתים קרובות במשפטי **while** ו-**do**, שנלמד עליהם בפרק 5. אם אתה משתמש בצמד האופרטורים להוספה ולהפחתה באופן בלתי תלוי, רצוי להשתמש במצב postfix ולהיות עקבי.

☯ **אם ברצונך להמשיך לפרק הבא**

השאר את סביבת הפיתוח Visual Studio 2005 פעילה ועבור לפרק 3.

☯ **אם ברצונך לסגור את Visual Studio 2005**

בתפריט File בחר Exit. כאשר תופיע תיבת דו-שיח Save, לחץ Yes כדי לשמור את עבודתך.

פרק 2 – טבלה מסכמת

המשימה	צריך
להכריז על משתנה	לכתוב את השם של סוג הנתון, לאחריו את שם המשתנה ולבסוף נקודה-פסיק. לדוגמה: `int outcome;`
לשנות את ערך המשתנה	לכתוב את ערך המשתנה מצד שמאל, לאחריו את אופרטור ההצבה (assignment), אחריו ביטוי המחשב את הערך החדש, ולבסוף נקודה-פסיק. לדוגמה: `outcome = 42;`
להמיר מחרוזת (string) לשלם (int)	לקרוא לשיטה **System.Int32.Parse**. לדוגמה: `System.Int32.Parse("42");`
לעקוף קדימות (precedence)	לשלב סוגריים בביטוי כדי לאלץ את האופרנדים לחבור לאופרטור הרצוי. לדוגמה: `(3 + 4) * 5`
להגדיל או להקטין משתנה	להשתמש באופרטור ++ או - -. לדוגמה: `count++;`

פרק 3

כתיבת שיטות והגדרת תחומי הכרזה

בסיום פרק זה, תוכל:

- להכריז על שיטות (methods) ולקרוא להן.
- להעביר נתון או ערך אל השיטה.
- לקבל ערך מהשיטה.
- להגדיר תחומי הכרזה מקומיים ומחלקתיים (local and class scope).
- להשתמש במהדר המשולב (integrated debugger) כדי להיכנס ולצאת משיטות בעודן פועלות.

בפרק 2 למדת להכריז על משתנים (variables), ליצור ביטויים (expressions) בעזרת אופרטורים (operators), ולשלוט בקדימות (precedence) ובשיוך (associativity) בביטויים המכילים מספר אופרטורים. בפרק זה תלמד על **שיטות** (methods). תלמד גם כיצד להשתמש בארגומנטים (arguments) ובפרמטרים (parameters) כדי להעביר מידע לשיטה וכיצד לקבל מידע חזרה מהשיטה באמצעות משפטי החזרה (return). לבסוף תלמד להיכנס ולצאת משיטות באמצעות המהדר המשולב (integrated debugger) של Windows Visual Studio 2005. הדבר ישמש אותך כאשר תרצה לעקוב אחר הביצוע של השיטה שיצרת במקרים בהם אין היא פועלת כפי שתכננת.

הכרזה על שיטה

שיטה (**method**) היא רצף של משפטים. אם עבדת בעבר בתוכנות כגון C או Visual Basic, תמצא ששיטה דומה מאוד ל-function או subroutine. לכל שיטה יש שם וגוף. עליך לבחור לשיטה שם בעל משמעות המעיד על מטרתה הכללית (**CalculateIncomeTax**, למשל). גוף השיטה מכיל את המשפטים אשר מופעלים כאשר קוראים לה. רוב השיטות יכולות לקבל נתונים לעיבוד ולהחזיר מידע, שלרוב יהיה תוצאת העיבוד. שיטות הן נדבך חשוב ומרכזי בתכנות.

התחביר להכרזה על שיטה

התחביר להגדרת שיטה בשפה Microsoft Visual C# נראה כך:

```
returnType methodName( parameterList )
{
   // method body statements go here
}
```

טיפ

זכור שכל הטקסט שכתוב לאחר לוכסן כפול ועד הקשת Enter נחשב להערה מבלי להתייחס לתוכנו.

- **returnType** הוא שם סוג הערך המוחזר, ומציין את סוג הערך שהשיטה תחזיר. הוא יכול להיות שם של כל סוג נתון, כגון **int** או **string**. אם השיטה אינה מחזירה ערך, עליך להשתמש במילת המפתח **void** במקום סוג הערך המוחזר (נראה דוגמה לכך בהמשך).
- **methodName** הוא השם שמשמש לקרוא לשיטה. על שמות השיטות לעמוד באותן הדרישות של שמות המשתנים. לדוגמה, **addValues** הוא שם חוקי לשיטה, אולם **add$Values** אינו חוקי. בשלב זה השתמש בשיטת הסימון camelCase לשמות השיטות, ורצוי גם שהמילה הראשונה תהיה פועל, לדוגמה - **displayCustomer**.
- **parameterList** הוא רשות ונועד לפרט את הסוגים והשמות השונים של הערכים, הנתונים שניתן להעביר לשיטה. את הפרמטרים עליך לכתוב בין הסוגריים בדומה להכרזת משתנים, כאשר שם הפרמטר יופיע מייד לאחר סוג הפרמטר. אם השיטה כוללת שני פרמטרים או יותר, עליך להפרידם בפסיקים.
- המשפטים שבגוף השיטה הם שורות הקוד המופעלות בעת הקריאה לשיטה. הם מתוחמים על ידי צמד סוגריים מסולסלים { }.

חשוב

מתכנתים מרקע תכנות בשפות C, ++C ו- Visual Basic יבחינו שבשפת #C אין תמיכה בשיטות גלובליות. עליך לכתוב את כל השיטות במסגרת של מחלקה, אחרת הן לא יעברו הידור.

לדוגמה, נגדיר שיטה בשם **addValues** בעלת שני פרמטרים מסוג **int** בשם **leftHandSide** ו-**rightHandSide**, אשר מחזירה תוצאה מסוג **int**:

```
int addValues(int leftHandSide, int rightHandside)
{
   // ...
   // method body statements go here
   // ...
}
```

להלן הגדרת שיטה בשם **showResult** בעלת פרמטר אחד **answer** מסוג **int**, אשר אינה מחזירה תוצאה:

```
void showResult(int answer)
{
   // ...
}
```

שים לב לשימוש במילת המפתח **void** נועדה לציין שהשיטה אינה מחזירה ערך.

חשוב

מתכנתים מרקע תכנות בשפה Visual Basic יבחינו ששפת #C אינה משתמשת במילות מפתח שונות, כדי לציין שיטות שמחזירות ערך (function) ושיטות שאינן מחזירות ערך (procedure או subroutine). עליך לציין תמיד את סוג הערך המוחזר, או לכתוב **void**.

משפטי החזרה

אם ברצונך לבנות שיטה המחזירה ערך או תוצאה (במילים אחרות, שיטה שסוג הערך המוחזר שלה אינו **void**), עליך לכלול בה **משפט החזרה** (**return statement**). יש לעשות זאת באמצעות מילת המפתח **return** שלאחריה ביטוי אשר מחשב את הערך המוחזר ולסיים בנקודה-פסיק. סוג הביטוי חייב להיות זהה לביטוי שצוין בעת הגדרת השיטה. במילים אחרות, אם השיטה מחזירה ערך **int**, משפט ההחזרה חייב גם הוא להחזיר ערך **int**. אם תנאי זה לא יתקיים, התוכנית לא תוכל לעבור הידור. להלן דוגמה:

```
int addValues(int leftHandSide, int rightHandside)
{
   // ...
   return leftHandSide + rightHandside;
}
```

על משפט **ההחזרה** להופיע בסוף השיטה, מכיוון שהוא גורם לסיומה וליציאה ממנה. משפטים שישנם לאחר משפט **ההחזרה**, לא יבוצעו (במקרה שכזה תקבל אזהרה מהמהדר).

אם אינך רוצה שהשיטה תחזיר ערך (או במילים אחרות, סוג הערך המוחזר הוא **void**), באפשרותך להשתמש בגרסה שונה של משפט ההחזרה אשר תגרום לסיום פעולת השיטה ויציאה ממנה. הקלד את מילת המפתח **return** ומייד אחריה נקודה-פסיק (עליך להשתמש במילת המפתח void גם במקרה זה), כמו לדוגמה:

```
void showResult(int answer)
{
   // display the answer
   // ...
   return;
}
```

אם השיטה אינה מחזירה ערך, אפשר לוותר על משפט ההחזרה מכיוון שהיא מסתיימת באופן אוטומטי בסוגר המסולסל הימני שבסופה. למרות שנתקלים לעיתים קרובות בהשמטת משפט ההחזרה, אין זה נחשב לנוהג טוב מבחינה סגנון תכנות ורצוי לכלול אותו תמיד.

בתרגיל הבא נבחן גרסה נוספת של היישום MathsOperators מפרק 2. גרסה זו שופרה על ידי שימוש דייקני במספר שיטות קטנות.

טיפ

אין גודל מינימלי לשיטה. אם השיטה עוזרת במניעת חזרות ומקלה על הבנת התוכנית שלך, הרי שהיא שימושית ואין חשיבות לגודלה.

בדומה, אין גודל מקסימלי לשיטה, אולם רצוי לכתוב קטע קוד קצר ככל הניתן. אם מספר הפקודות של השיטה גולש מגודל המסך, רצוי לשקול פירוק למספר שיטות קטנות, מכיוון שקוד קצר הינו קוד קריא, קל להבנה ופעולת ניפוי השגיאות פשוטה יותר.

בחן את הגדרות השיטה

1. הפעל את Visual Studio 2005.
2. פתח את הפרויקט Methods שבתיקייה \My Documents\Microsoft Press\Visual CSharp Step by Step\Chapter 3\Methods.
3. בתפריט Debug בחר Start Without Debugging.

   סביבת הפיתוח Visual Studio 2005 תבנה ותפעיל את היישום.
4. השתמש ביישום כדי להיזכר באופן פעולתו, ולאחר מכן לחץ Quit.
5. הצג את הקוד עבור Form1.cs בחלון Code and Text Editor (ב- Solution Explorer, לחץ לחיצה-ימנית על Form1.cs ובתפריט הקיצור בחר View Code).
6. בחלון Code and Text Editor, אתר את השיטה addValues.

   השיטה נראית כך:

```
private int addValues(int leftHandSide, int rightHandSide)
{
   expression.Text = leftHandSide.ToString() + " + " +
    rightHandSide.ToString();
   return leftHandSide + rightHandSide;
}
```

השיטה **addValues** כוללת שני משפטים. המשפט הראשון מציג את החישוב המבוצע בתיבת הטקסט **expression** אשר בטופס. ערכי הפרמטרים **leftHandSide** ו-**rightHandSide** מומרים למחרוזות (באמצעות השיטה **ToString** שראינו בפרק 2) המשורשרות יחדיו עם הסימן "+" ביניהן.

המשפט השני מחבר את ערכי שני המשתנים מסוג **int**, **leftHandSide** ו-**rightHandSide**, באמצעות האופרטור + ומחזיר את תוצאת החיבור. זכור שחיבור שני משתני int יוצר משתנה **int** נוסף, ולכן סוג הערך המוחזר של **addValues** הוא **Int**.

7. בחלון Code and Text Editor, אתר את השיטה **ShowResult**.

   השיטה תיראה כך:

```
private void showResult(int answer)
{
   result.Text = answer.ToString();
}
```

השיטה כוללת משפט אחד המציג מחרוזת המייצגת את הפרמטר **answer** שבתיבת הטקסט **result**.

קריאה לשיטות

שיטות קיימות כדי לקרוא להן. קוראים לשיטה בשמה כדי שתבצע את משימתה. אם השיטה מקבלת ערך או ערכים (על פי הפירוט בפרמטרים שלה), עליך לספק אותם כנדרש. אם השיטה מחזירה ערך (על פי המפורט בה), עליך לשמור אותו לשימוש מיידי או מאוחר.

תחביר הקריאה לשיטות

התחביר של שפת #C המשמש לקריאה לשיטות הוא:

```
methodName ( argumentList )
```

- **methodName** (שם השיטה) חייב להיות זהה לחלוטין לשם השיטה שהוא קורא. זכור ששפת #C מבחינה בין אותיות גדולות וקטנות.
- **argumentList** מייצג מידע שהוא רשות. בדרך כלל אלה הם ערכים שהשיטה מקבלת. עליך להזין ארגומנט עבור כל פרמטר, וערכו חייב להתאים לסוג הפרמטר הרלוונטי. אם המחלקה שאתה קורא כוללת שני פרמטרים או יותר, עליך להפריד ביניהם באמצעות פסיקים (,).

חשוב

עליך לכלול את הסוגריים בכל קריאה לשיטה, גם כאשר אין לה ארגומנטים.

הנה שוב השיטה **addValues**:

```
int addValues(int leftHandSide, int rightHandside)
{
   // ...
}
```

לשיטה **addValues** יש שני פרמטרים מסוג **int**, לכן עליך לקרוא לה על ידי שני ארגומנטים מסוג **int** שביניהם פסיק להפרדה:

```
addValues(39, 3);          // okay
```

ניתן גם להחליף את הערכים 39 ו–3 בשני משתנים מסוג **int**. הערכים המאוחסנים במשתנים יועברו לשיטה בתור ארגומנטים, הנה כך:

```
int arg1 = 99;
int arg2 = 1;
addValues(arg1, arg2);
```

אם תנסה לקרוא לשיטה **addValues** בכל דרך אחרת, קרוב לוודאי שהניסיון ייכשל. הסיבות לכישלון מוסברות בדוגמאות הבאות:

```
addValues;            //compile time error, no parentheses
addValues();          //compile time error, not enough arguments
addValues(39);        //compile time error, not enough arguments
addValues("39", "3"); //compile time error, wrong types
```

השיטה **addValues** מחזירה ערך מסוג **int**, אשר יכול לשמש בכל מקום שבו מותר ערך **int** רגיל. בחן את הדוגמאות שלהלן:

```
result = addValues(39, 3); // on right hand side of an assignment
```

המשתנה **result** יקבל את הערך המוחזר מהשיטה addValues

```
showResult(addValues(39, 3)); // as argument to another method call
```

השיטה showResult תשתמש בערך המוחזר מ- addValues כארגומנט לתוכה.

גם התרגיל הבא משתמש ביישום MathsOperators. נבחן מספר קריאות לשיטות המופיעות בו.

בחן קריאות לשיטות

1. חזור לפרויקט Methods (הפרויקט כבר פתוח בסביבת הפיתוח אם אתה ממשיך מהתרגיל הקודם. אם לא, פתח אותו מתוך התיקייה \My Documents\Microsoft Press\Visual CSharp Step by Step\Chapter 3\Methods).
2. בחלון Code and Text Editor הצג את הקוד עבור Form1.cs.
3. אתר את השיטה **calculate_Click**, והבט על שני המשפטים הראשונים בה לאחר המשפט **try** והסוגר המסולסל השמאלי. (הסבר נוסף על משפטי try ניתן בפרק 6).

   המשפטים הם:

```
int leftHandSide = System.Int32.Parse(leftHandSideOperand.Text);
int rightHandSide = System.Int32.Parse(rightHandSideOperand.Text);
```

   שני המשפטים מכריזים על שני משתני int בשמות **leftHandSide** ו-**rightHandSide**. מה שמעניין אותנו במשפטים אלה הוא האופן שבו הם מאותחלים. עבור שני המשתנים מופעלת השיטה **Parse** מהמחלקה **System.Int32** (להזכירך, **System** הוא מרחב השמות ו-**Int32** הוא שם המחלקה שבמרחב שמות זה). שיטה זו מקבלת פרמטר **string** יחיד וממירה אותו לערך מסוג **int**. שתי שורות הקוד הללו לוקחות את המידע שהקליד המשתמש בטופס לתיבות הטקסט **leftHandSideOperand** ו-**rightHandSideOperand**, וממיר אותם לערכי **int**.
4. הבט במשפט הרביעי בשיטה **calculate_Click** (לאחר המשפט **if** וסוגר מסולסל שמאלי נוסף):

```
calculatedValue = addValues(leftHandSide, rightHandSide);
```

משפט זה קורא לשיטה **addValues** ומעביר לה כארגומנטים את הערכים שבמשתנים **leftHandSide** ו-**rightHandSide**. הערך שמוחזר מהשיטה **addValues** מאוחסן במשתנה **calculateValue**.

5. הבט במשפט הבא:

```
showResult(calculatedValue);
```

משפט זה קורא לשיטה **showResult** ומעביר לה כארגומנט את הערך שבמשתנה **calculateValue**. השיטה **showResult** אינה מחזירה ערך.

6. בחלון Code and Text Editor, אתר את השיטה **showResult** שבחנת קודם לכן (תוכל לסמן את שם השיטה ולהקיש F12 כדי להגיע אל שורת הגדרת הפונקציה).

השיטה כוללת משפט אחד בלבד:

```
result.Text = answer.ToString();
```

שים לב לסוגריים המופיעים בקריאה לשיטה **ToString**, למרות היעדר הארגומנטים. חובה לכלול אותם תמיד.

טיפ

ניתן לקרוא לשיטות השייכות לאובייקטים אחרים על ידי הוספת שם האובייקט לפני השיטה. בדוגמה הקודמת, הביטוי **answer.ToString()** קרא לשיטה **ToString** השייכת לאובייקט **answer**.

הגדרת תחום הכרזה

במספר דוגמאות ראית שניתן ליצור משתנה בתוך שיטה. המשתנה נוצר במקום שהמשפט נמצא בו. המשפטים שמופיעים **בשיטה** לאחריו יכולים להתבסס על כך שכבר הוגדר. במילים אחרות, ניתן להשתמש במשתנה רק לאחר יצירתו, ועם סיום השיטה לא ניתן להשתמש בו עוד. כלומר, אורך חייו כאורך חיי השיטה.

כאשר משתנה יכול לשמש באזור מסוים בתוכנית, נאמר שזה **תחום ההכרזה** (**scope**) שלו. במילים אחרות, תחום ההכרזה של משתנה הוא חלק התוכנית שבו מתאפשר השימוש בו. בדומה למשתנים, תחום הכרזה תקף גם לשיטות. תחום ההכרזה של מזהה (identifier) של משתנה או שיטה, מוגדר על פי המקום בתוכנית שבו הוכרז, כפי שתלמד כעת.

הגדרת תחום הכרזה מקומי

הסוגריים המסולסלים השמאליים והימניים היוצרים את גוף השיטה יוצרים גם תחום הכרזה. תחום ההכרזה של כל משתנה אשר יוכרז בגוף המחלקה נקרא משתנה מקומי (**local variable**), אשר מוכר וניתן לשימוש בתוך השיטה בלבד ולא ניתן לגשת אליו משיטות אחרות במחלקה.

בדוגמה הבאה המשתנה myVar מוגדר בשיטה firstMethod. זהו משתנה מקומי ותחום הכרזתו הוא בשיטה זו בלבד.

```
class Example
{
    void firstMethod()
    {
        int myVar;
        ...
    }

    void anotherMethod()
    {
        myVar = 42; // error - variable not in scope
        ...
    }
}
```

כל ניסיון לקרוא למשתנה הזה מחוץ לשיטה זו יניב שגיאת הידור.

הגדרת תחום הכרזה ברמת המחלקה

הסוגריים המסולסלים השמאליים והימניים היוצרים את גוף המחלקה יוצרים גם תחום הכרזה. תחום ההכרזה של כל משתנה אשר יוכרז בגוף המחלקה, אך מחוץ לשיטה, יהיה בכל המחלקה. השם הנכון למשתנה שהוגדר במסגרת של מחלקה הוא **שדה** (**field**), או **משתנה מחלקה**. בניגוד למשתנים מקומיים, השדות מאפשרים לשיטות שבמחלקה לחלוק ביניהם את הערכים שלהם. להלן דוגמה:

```
class Example
{
    void firstMethod()
    {
        myField = 42;   // ok
        ...
    }

    void anotherMethod()
    {
        myField = 42;   // ok
        ...
    }

    int myField = 0;
}
```

המשתנה **myField** מוגדר במסגרת המחלקה, אולם מחוץ לשיטות **firstMethod** ו-**anotherMethod**. לפיכך, ל-**myField** יש תחום הכרזה מחלקתי וכל השיטות במחלקה יכולות לעשות בו שימוש.

שים לב לדבר נוסף. במסגרת שיטה עליך להגדיר משתנה לפני שתוכל להשתמש בו. שדות שונים מבחינה זו, בכך ששיטה יכולה לפנות לשדה (משתנה מחלקה) גם לפני הופעת המשפט המכריז עליו. המהדר פותר עבורך את בעיית סדר הכתיבה הכרונולוגי.

העמסת שיטות

כאשר שני מזהים (identifiers) הם בעלי אותו שם ומוכרזים באותו תחום, הם נחשבים **למועמסים** (**overloaded**). מזהים מועמסים נחשבים לעיתים קרובות לבאג, ומוצגים כשגיאה בזמן ההידור. לדוגמה, אם תכריז באותה שיטה על שני משתנים מקומיים בעלי אותו שם, תופיע הודעת שגיאה. כמו-כן, אם תגדיר שני שדות בעלי אותו שם באותה מחלקה או שני שדות זהים באותה מחלקה, תתקבל שגיאת הידור. לכאורה נראה שכל פעם שנבצע פעולה כזו תתקבל שגיאה, אולם יש דרך להעמיס מזהים בצורה יעילה גם מועילה.

היזכר בשיטה **WriteLine** מהמחלקה **Console**, שבה השתמשת כדי להציג (להדפיס) משפט על המסך. כאשר אתה מקליד **WriteLine** בחלון Code and Text Editor בעת כתיבת קוד #C, רשימת IntelliSense נותנת לך 19 אפשרויות שונות! כל אחת מהגרסאות של **WriteLine** מקבלת פרמטרים אחרים. גרסה אחת אינה מקבלת פרמטרים ומדפיסה שורה ריקה, אחרת מקבלת פרמטר מסוג bool ומדפיסה את ייצוג הערך שלו (**true** או **false**), גרסה נוספת מקבלת פרמטר דצימלי ומדפיסה אותו כמחרוזת וכו'. בזמן ההידור, המהדר בוחן את סוג הפרמטרים שאתה מציב בשיטה, וקורא לשיטה המתאימה. להלן דוגמה:

```
static void Main()
{
   Console.WriteLine("The answer is ");
   Console.WriteLine(42);
}
```

העמסה (overloading) שימושית בעיקר במקרים שבהם עליך לבצע פעולה זהה על סוגי נתונים שונים. כך, ניתן להעמיס שיטה אשר צריכה לבצע משימות שונות על פי סוגים שונים של פרמטרים. אפשרות זו קיימת כדי שתוכל בעת הקריאה לשיטה, להציג רשימת פרמטרים המופרדים בפסיק ולפיהם המהדר ידע לבחור באחת מהשיטות המועמסות. אולם זכור, למרות שניתן להעמיס את הפרמטרים המתקבלים על ידי השיטה, לא ניתן להעמיס את סוג הערך המוחזר של השיטה. כלומר, לא ניתן להכריז על שתי שיטות בעלות אותו שם, אשר מובדלות רק על ידי סוג הערך שהן מחזירות (המהדר חכם, אבל לא עד כדי כך). רשימת הפרמטרים המועברים לשיטה נקראת גם **חתימת השיטה** (חתימת הפונקציה). חתימת השיטה מגדירה עבור צרכני השיטה, מה היא מצפה לקבל כארגומנטים.

כתיבת שיטות

בתרגילים הבאים עליך ליצור שיטה המחשבת כמה יועץ מקצועי אמור לגבות מהלקוח על מספר ימי ייעוץ בתעריף יומי קבוע. תחילה עליך לפתח את הלוגיקה המשמשת את היישום, ואחר כך תיעזר באשף Generate Method Stub כדי לכתוב שיטות על פי אותה הלוגיקה. בשלב הבא תפעיל את השיטות ביישום console כדי לקבל תחושה של ביצועי התוכנית ואפשרויותיה. לבסוף, תיכנס ותצא מהשיטות בזמן שהן פועלות, באמצעות המנפה (debugger) של Visual Studio 2005.

פיתוח התוכנית

1. בסביבת הפיתוח Visual Studio 2005, פתח את הפרויקט DailyRate שבתיקייה .\My Documents\Microsoft Press\Visual CSharp Step by Step\Chapter 3\DailyRate.
2. ב- Solution Explorer, לחץ לחיצה-כפולה על הקובץ Program.cs כדי להציג את קוד התוכנית בחלון Code and Text Editor.
3. הוסף את המשפטים הבאים לגוף השיטה run:

```
double dailyRate = readDouble("Enter your daily rate: ");
int noOfDays = readInt("Enter the number of days: ");
writeFee(calculateFee(dailyRate, noOfDays));
```

השיטה **Main** קוראת לשיטה **run** בעת הפעלת היישום (כדי להבין את מנגנון הפעולה, המתן עד פרק 7, הדן במחלקות).

בלוק הקוד שהוספת לשיטה **run** קורא לשיטה **readDouble** (אשר תכתוב מייד) אשר מבקשת מהמשתמש להזין את התעריף היומי של היועץ. המשפט הבא קורא לשיטה **readInt** (גם אותה תכתוב) כדי לקבל את מספר הימים. המשפט האחרון קורא לשיטה **writeFee** (תיכתב גם היא) המציגה את התוצאה על המסך. שים לב שהערך המועבר לשיטה **writeFee** הוא הערך שהוחזר על ידי השיטה **calculateFee** (השיטה האחרונה שעליך לכתוב), אשר מקבלת כפרמטרים את התעריף היומי ואת מספר הימים ומחשבת את הסכום הכולל לתשלום.

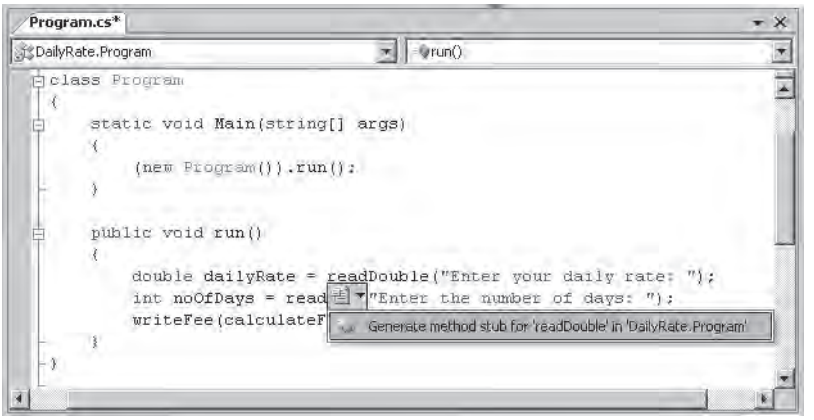
הערה

מכיוון שטרם כתבת את השיטות **readDouble**, **readInt**, **writeFee** או **calculateFee**, הרשימה IntelliSense לא תכלול אותן בעת הקלדת קוד זה. מאותה סיבה, אל תנסה להדר את היישום כי הניסיון ייכשל.

כתוב את השיטות בעזרת האשף Generate Method Stub

1. בחלון Code and Text Editor, לחץ על הקריאה לשיטה **readDouble** שנמצאת בתוך השיטה **run**.

קו תחתי קטן יופיע מתחת לאות הראשונה ("r") של readDouble. מיקום סמן העכבר מעל האות "r" יגרום להופעת סמל. אם תזיז את סמן העכבר לסמל תופיע ההודעה "Options to generate a method stub (Shift + Alt + F10)" ולידה תפריט נפתח. לחץ עליו ותופיע האפשרות "Generate method stub for 'readDouble' in 'DailyRate.Program'", כלהלן:



2. לחץ על Generate method stub for 'readDouble' in 'DailyRate.Program'.

כתוצאה, האשף Generate Method Stub יבחן את הקריאה לשיטה **readDouble**, יאמת את סוג הפרמטרים והערך המוחזר של השיטה, והשיטה readDouble תיווצר ובתוכה תראה את שורת ברירת מחדל:

```
private double readDouble(string p)
{
 throw new Exception ("The method or operation is not implemented.");
}
```

השיטה החדשה מסווגת כפרטית (**private**), שמשמעויותיו יוסברו בפרק 7. גוף השיטה מטיל חריג (throw an **exception**, פעולה שתוסבר בפרק 6). בשלב הבא עליך להחליף את גוף השיטה עם קוד משלך.

3. מחק את המשפט **throw new Exception(...);** מהשיטה **readDouble** והחלף אותו עם שורות הקוד הבאות:

```
Console.Write(p);
string line = Console.ReadLine();
return double.Parse(line);
```

בלוק קוד זה מציג במסך את המחרוזת המאוחסנת במשתנה **p**. משתנה **p** הוא פרמטר מסוג string שהועבר לשיטה כאשר נקראה, והוא מכיל הודעה המבקשת מהמשתמש להקליד את התעריף היומי. המשתמש מקליד באמצעות השיטה **ReadLine** ערך המתקבל כ-**string** ומומר ל-**double** על ידי השיטה **double.Parse**. התוצאה המתקבלת מהווה את הערך המוחזר כתוצאה מהקריאה לשיטה.

הערה

השיטה **ReadLine** היא אחות לשיטה **WriteLine**. היא קוראת קלט שמזין המשתמש באמצעות המקלדת, ומסתיימת כאשר המשתמש מקיש **Enter**. הטקסט שהקליד המשתמש הוא הערך המוחזר של שיטה זו.

4. לחץ על הקריאה לשיטה **readInt** שבתוך השיטה **run**, והשתמש באותו תהליך מוקדם, כדי ליצור תבנית לשיטה **readInt**.

השיטה **readInt** תיווצר עם פעולת ברירת מחדל שכלולה בה.

טיפ

יש דרך נוספת ליצירת תבנית שיטה. לחץ לחיצה ימנית על הקריאה לשיטה, ובתפריט הקיצור בחר Generate Method Stub.

5. החלף את גוף השיטה readInt עם המשפטים הבאים:

```
Console.Write(p);
string line = Console.ReadLine();
return int.Parse(line);
```

בלוק קוד זה דומה מאוד לזה שהצבנו בשיטה **readDouble**. ההבדל היחיד הוא שהשיטה מחזירה ערך **int**, ולכן המחרוזת מומרת למספר באמצעות השיטה **int.Parse**.

6. לחץ לחיצה ימנית על הקריאה לשיטה **calculateFee** שבתוך השיטה **run**, ומתפריט הקיצור בחר Generate Method Stub.

כתוצאה תיווצר השיטה **calculateFee**.

```
private object calculateFee(double dailyRate, int noOfDays)
{
 throw new Exception("The method or operation is not implemented.");
}
```

שים לב שהאשף Generate Method Stub משתמש בשמות הארגומנטים כדי ליצור שמות לפרמטרים (כמובן שאתה יכול לשנות שמות אלה אם אינם לטעמך). מעניין יותר הוא סוג הערך המוחזר על ידי השיטה **object**. האשף Generate Method Stub אינו יכול לקבוע בוודאות את סוג הערך המוחזר על ידי השיטה על פי האופן שבו נקראה. משמעות הסוג **object** היא "משהו (thing)" כללי, ועליך לשנות אותו לסוג הנחוץ לך בעת כתיבת הקוד עבור השיטה. בפרק 7 נדון בהרחבה על הסוג **object**.

7. שנה את הגדרת השיטה **calculateFee** כדי שתחזיר ערך מסוג **double**:

```
private double calculateFee(double dailyRate, int noOfDays)
{
 throw new Exception("The method or operation is not implemented.");
}
```

8. החלף את גוף השיטה **calculateFee** עם המשפט הבא, אשר מחשב את הסכום לתשלום על ידי כפל שני הפרמטרים והחזרת התוצאה:

```
return dailyRate * noOfDays;
```

9. לחץ לחיצה ימנית על הקריאה לשיטה **writeFee** אשר בשיטה **run**, ומתפריט הקיצור בחר Generate Method Stub.

כתוצאה תיווצר השיטה **writeFee**. שים לב שהאשף Generate Method Stub משתמש בהגדרות השיטה **calculateFee** כדי לקבוע שהפרמטר צריך להיות מסוג **double**. כמו כן, מכיוון שהשיטה אינה מחזירה ערך, הוגדר הסוג **void**:

```
private void writeFee(double p)
{
   ...
}
```

10. הקלד את המשפט הבא בגוף השיטה **writeFee**:

```
Console.WriteLine("The consultant's fee is: {0}", p * 1.1);
```

הערה

גרסה זו של השיטה **WriteLine** מדגימה את השימוש במחרוזת בפורמט פשוט (simple format string). הטקסט **{0}** מציין ציון מיקום (placeholder) המוחלף עם ערך הביטוי שאחריו **(p * 1.1)** בזמן ההפעלה.

11. בתפריט Build בחר Build Solutions.

טיפ

אם לדעתך אתה שולט מספיק בתחביר, תוכל להקליד שיטות בחלון Code and Text Editor ולא באמצעות האשף Generate Method Stub.

Refactoring Code

מאפיין שימושי ביותר של Visual Studio 2005 היא היכולת לבצע Refactor (ליכוד מחדש) לקוד. מתכנת מקליד קוד אשר ידוע לו מראש שאותו בלוק קוד ישמש אותו גם במקומות נוספים ביישום. במקרה זה הוא רוצה שהקוד יהיה נגיש לשימוש חוזר. אם לפניך מקרה כזה, סמן את בלוק הקוד, ובתפריט Refactor בחר Extract Method. בתיבת הדו-שיח Extract Method אשר תופיע על המסך, תתבקש לקבוע שם לשיטה החדשה. הקלד שם ולאחר מכן לחץ OK, כדי ליצור שיטה חדשה המכילה את הקוד המסומן, והקוד שסימנת יוחלף במשפט הקורא לשיטה החדשה. הפקודה Extract Method מחוכמת דיה כדי לקבוע אם השיטה אמורה לקבל פרמטרים ולהחזיר ערך. מכל מקום בתוכנית תוכל לקרוא לשיטה החדשה.

בדוק את התוכנית

1. בתפריט Debug בחר Start Without Debugging.

   Visual Studio 2005 תבנה ותפעיל את התוכנה, וחלון תצוגה יופיע.

2. הקלד 525 ב- Enter Your Daily Rate, והקש Enter.

3. הקלד 17 ב- The Number of Days, והקש Enter.

   התוכנית תדפיס את ההודעה הבאה בחלון התצוגה:

```
The consultant's fee is: 9817.5
```

4. הקש Enter כדי לחזור לסביבת הפיתוח של Visual Studio 2005.

בתרגיל האחרון לפרק זה תשתמש במנפה (debugger) של Visual Studio 2005 כדי להפעיל את התוכנית בהילוך איטי ולצפות בה. הכוונה היא שתוכל לעקוב אחר שלבי הביצוע של התוכנית דרך השיטות השונות מתחילת התוכנית ועד סופה. תוכל לצפות כיצד נעשית כניסה לשיטה (**stepping into the methods**) ותראה כיצד כל משפט החזרת ערך גורם ליציאה משיטה (**stepping out of the method**). בעת כניסה ויציאה משיטות, תשתמש בכלים שבסרגל הכלים Debug. אולם תוכל למצוא את הפקודות הללו גם בתפריט Debug כאשר התוכנית פועלת במצב זה.

צעד דרך השיטות באמצעות המנפה של Visual Studio 2005

1. אתר את השיטה **run** בחלון Code and Text Editor.

2. הצב את סמן העכבר על המשפט הראשון בשיטה **run**, אשר נראה כך:

```
double dailyRate = readDouble("Enter your daily rate: ");
```

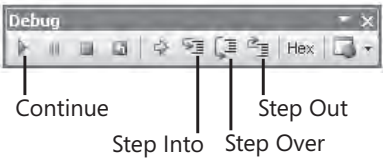
3. לחץ לחיצה ימנית בעכבר, ובתפריט הקיצור בחר Run To Cursor.

   התוכנית תפעל עד שתגיע למשפט הראשון בשיטה **run**, ותעצור. חץ צהוב בשול השמאלי של החלון Code and Text Editor יסמן את המשפט הנוכחי, אשר מסומן גם הוא ברקע צהוב.

4. בתפריט View, Toolbars ודא שמסומן סרגל הכלים Debug.

   אם סרגל הכלים Debug לא היה מסומן, הוא יוצג כעת. ייתכן שהוא מחובר לסרגלי כלים אחרים. אם אינך מוצא אותו, נסה להסתיר את סרגל הכלים שוב. שים לב איזה לחצנים נעלמים מהתצוגה, והצג אותו שוב.

   סרגל הכלים Debug נראה כך:



טיפ

כדי להציג את סרגל הכלים Debug בחלון נפרד, גרור אותו מהסימון שבצידו השמאלי, אל החלון Code and Text Editor.

5. בסרגל הכלים Debug לחץ Step Into.

הפעולה תגרום למנפה להיכנס לשיטה שקוראים לה. הסמן הצהוב יעבור לסוגר המסולסל השמאלי שבתחילת השיטה **readDouble**. לחץ Step Into בשנית. הסמן יתקדם אל המשפט הראשון:

```
Console.Write(p);
```

טיפ

מקש הקיצור ללחצן Step Into הוא F11.

לחץ שוב Step Into הסמן הצהוב יעבור למשפט השני בשיטה, והתוכנית תציג את הבקשה Enter Your Daily Rate בחלון התצוגה (Console window) הקלד 525 והקש Enter כדי לחזור ל- Visual Studio 2005 (ייתכן שחלון התצוגה יוסתר מאחורי חלון Visual Studio). המשך את ריצת התוכנית באמצעות Step Into עד ליציאה מהשיטה readDouble, ועצור כאשר השורה הבאה מסומנת (בשיטה run):

```
int noOfDays = readInt("Enter the number of days: ");
```

6. בסרגל הכלים Debug לחץ Step Over.

הפעולה תגרום לשיטה **readInt** לפעול מבלי שתיכנס לתוכה. כעת, הסמן הצהוב ייעלם וחלון התצוגה יעבור לקדמת המסך, כי התוכנית מבצעת את השיטה Console.ReadLine, ועליך להקליד ערך עבור מספר הימים.

7. הקלד 17 בחלון התצוגה והקש Enter.

השליטה תחזור לסביבת הפיתוח. הסמן הצהוב יעבור למשפט השלישי בשיטה **run**.

טיפ

F10 הוא מקש הקיצור אל הלחצן Step Over.

מבלי ללחוץ, הזז את סמן העכבר אל המשתנה dailyRate בשורה הראשונה, או אל noOfDays בשורה השנייה. מסגרת קטנה המציגה את הערך הנוכחי של המשתנה תופיע על המסך. ניתן להשתמש במאפיין זה כדי לוודא שמשתנה נקבע לפי הערך המצופה בעת כניסה ויציאה משיטות.

8. בסרגל הכלים Debug לחץ Step Into.

הסמן הצהוב יעבור אל הסוגר המסולסל השמאלי שבתחילת השיטה calculateFee. שיטה זו מחושבת ראשונה ונקראת לפני השיטה writeFee, מכיוון שהערך המוחזר מהשיטה calculateFee יתקבל כארגומנט לשיטה writeFee.

9. בסרגל הכלים Debug, לחץ Step Out. במקרה זה המנפה יעזוב את הבלוק שבו הוא נמצא, ויחזור אל המקום שממנו הגיע.

טיפ

מקש הקיצור ללחצן Step Out הוא Shift + F11.

10. הסמן הצהוב יחזור למשפט השלישי בשיטה **run**.

11. בסרגל הכלים Debug לחץ Step Into.

הפעם, הסמן הצהוב יעבור אל הסוגר המסולסל השמאלי שבתחילת השיטה writeFee.

12. הצב את סמן העכבר על המשתנה **p** שבהגדרת השיטה.

הערך 8925.0 של p, יוצג כעת.

13. בסרגל הכלים Debug, לחץ Step Out.

ההודעה The consultant's fee is: 9817.5 תופיע בחלון התצוגה (ייתכן שתצטרך להעביר את חלון התצוגה לקדמת המסך אם הוא מוסתר על ידי חלון סביבת הפיתוח Visual Studio 2005). הסמן הצהוב יחזור אל המשפט השלישי בשיטה run.

14. בסרגל הכלים Debug לחץ Continue, כדי לגרום לתוכנה להמשיך ולפעול מבלי לעצור בכל משפט.

טיפ

מקש הקיצור ללחצן Continue שבמנפה השגיאות הוא F5.

היישום יסיים את פעולתו.

יפה! כתבת שיטות וקראת להן, והשתמשת במנפה של Visual Studio 2005 כדי להיכנס ולצאת משיטות בעודן פועלות.

☯ **אם ברצונך להמשיך לפרק הבא**

השאר את Visual Studio 2005 פעילה, ועבור לפרק 4.

☯ **אם ברצונך לסגור את Visual Studio 2005**

בתפריט File בחר Exit. אם מוצגת תיבת דו-שיח Save, לחץ Yes כדי לשמור את עבודתך.

פרק 3 – טבלה מסכמת

המשימה	צריך

להכריז על שיטה	לכתוב את השיטה בתוך מחלקה. לדוגמה:
	`int addValues(int leftHandSide, int rightHandside)`
	`{`
	`  // ...`
	`}`

להחזיר ערך מתוך השיטה	לכתוב משפט החזרה (return) בתוך השיטה. לדוגמה: `return leftHandSide + rightHandside;`
להחזיר ערך מתוך השיטה לפני סיום השיטה	לכתוב משפט החזרה (nruter) בתוך השיטה. לדוגמה: `Return;`
לקרוא לשיטה	לרשום את שם השיטה ואת הארגומנטים הנחוצים בין שני סוגריים. לדוגמה: `addValues(39, 3);`
להשתמש באשף Generate Method Stub	לסמן משפט שקורא לשיטה. לחץ על ההודעה: Generate Method Stub (ניתן גם לקבל את ההודעה על ידי לחיצה על צירוף המקשים Alt + Shift + F10)
להציג את סרגל הכלים Debug	בתפריט View ← Toolbars בחר Debug.
להיכנס לשיטה (Step into a Method)	בסרגל הכלים Debug, לחץ Step Into. או בתפריט Debug לחץ Step Into. או הקש F11.
לצאת משיטה (Step out of a Method)	בסרגל הכלים Debug לחץ Step Out. או בתפריט Debug לחץ Step Out. או הקש Shift + F11.
לדלג מעל שיטה (Step Over)	בסרגל הכלים Debug, לחץ Step Over. או בתפריט Debug, לחץ Step Over. או הקש F10.

פרק 4

משפטי החלטה

בסיום פרק זה, תוכל:

- להכריז על משתנים מסוג **bool**.
- להשתמש באופרטורים בוליאניים כדי ליצור ביטויים שתוצאתם היא true (אמת) או false (שקר).
- לכתוב משפטי if (אם) לקבלת החלטות המבוססות על תוצאות הביטויים הבוליאניים.
- לכתוב משפטי switch (מיתוג) לקבלת החלטות מורכבות יותר.

בפרק 3 למדת לאחד מספר משפטים לשיטה. למדת גם כיצד להשתמש בארגומנטים ובפרמטרים כדי להעביר נתונים או ערכים לשיטות, וכיצד להשתמש במשפטי **החזרה**.

פיצול התוכנית למספר שיטות נפרדות, שכל אחת מהן מבצעת משימה או חישוב מסוים, היא דרך עבודה מומלצת. תוכניות רבות צריכות לפתור בעיות גדולות ומורכבות. פירוק התוכנית לשיטות מסייע להבנת הבעיות הללו ולהתמקד בנפרד בכל שלבי הפתרון, בזה אחר זה. עליך גם לדעת לכתוב שיטות המבצעות פעולות שונות לפי בחירה ובהתחשב בנסיבות. בפרק זה תלמד לבצע פעולות אלו.

הכרזה על משתנים מסוג bool

בעולם התכנות, בניגוד לעולם האמיתי, הכל שחור או לבן, נכון או לא נכון, אמת או שקר, ואין אזורים אפורים. לדוגמה, אם תיצור משתנה שלם (Integer variable) בשם x, תציב בו 99 ותשאל "האם x מכיל את הערך 99?", התשובה החד-משמעית תהיה אמת (true). אם תשאל "האם x קטן מ-10?", התשובה תהיה חד משמעית, שקר (false). אלו הן דוגמאות של **ביטויים בוליאניים** (**Boolean expressions**).

הערה

התשובות לשאלות אלו אינן חד-משמעיות בכל שפות התכנות. משתנה שלא הוצב בו ערך מכיל ערך בלתי מוגדר, ואינך יכול, לדוגמה, להגיד שהוא קטן מ-10 באופן חד-משמעי. זהו המקור הנפוץ ביותר לשגיאות בתוכניות C ו-C++. המהדר של Microsoft Visual C# פותר את הבעיה הזו בכך שהוא מוודא שתמיד תציב ערך במשתנה לפני השימוש בו. כאשר תנסה להשתמש בתוכן המשתנה לפני שהצבת בו ערך, התוכנית לא תהודר.

Microsoft Visual C# מכיל סוג נתון בשם **bool**. משתנה **bool** יכול להכיל אחד משני ערכים – **true** (אמת, נכון) או **false** (שקר, לא נכון). לדוגמה, שלושת המשפטים הבאים מכריזים על משתנה **bool** בשם **areYouReady**, מציבים בו אמת ומדפיסים את ערכו על המסך.

```
bool areYouReady;
areYouReady = true;
Console.WriteLine(areYouReady); // writes True
```

אופרטורים בוליאניים

אופרטור בוליאני הוא אופרטור שערכו **true** (אמת) או **false** (שקר). בשפת C# יש אופרטורים בוליאניים אחדים אשר שימושיים מאוד. הפשוט ביניהם הוא האופרטור NOT, המיוצג על ידי סימן קריאה (!). האופרטור **!** שולל ערך בוליאני ומניב את הערך ההפוך לו. בדוגמה האחרונה, אם ערכו של **areYouReady** הוא **אמת**, אזי ערכו של הביטוי **areYouReady!** יהיה **false**.

אופרטורים של השוואה ושל יחס

שני אופרטורים נפוצים ביותר הם **השוויון** (==) ו**אי-שוויון** (=!). תפקידם לבחון אם שני ערכים מאותו סוג הינם שווים זה לזה או לא. הטבלה הבאה מסכמת את פעולת שני האופרטורים בעזרת משתנה מסוג int בשם **age**, בתור דוגמה:

Operator	Meaning	Example	Outcome if age is 42
==	Equal to	age == 100	**false**
!=	Not equal to	age != 0	**true**

קבוצת אופרטורים נוספת היא אופרטורים של **יחס** (**relational**). אופרטורי היחס מאפשרים לדעת אם ערך כלשהו גדול או קטן מערך אחר מאותו סוג. הטבלה הבאה מסבירה את השימוש באופרטורים אלה.

Operator	Meaning	Example	Outcome if age is 42
<	Less than	age < 21	**false**
<=	Less than or equal to	age <= 18	**false**
>	Greater than	age > 16	**true**
>=	Greater than or equal to	age >= 30	**true**

הערה

אין לבלבל בין אופרטור ה**שוויון** == (סימן כפול) לבין אופרטור ה**השמה** = (סימן אחד). הקוד **x= =y** משווה בין x לבין y ומחזיר את הערך אמת אם הם שווים. לעומת זה, הפקודה x=y מציבה את הערך של y במשתנה x.

אופרטורים לוגיים מַתנים

שפת #C כוללת שני אופרטורים בוליאניים נוספים: האופרטור הלוגי **AND** (וגם) המיוצג על ידי הסמל &&, והאופרטור הלוגי **OR** (או) המיוצג על ידי הסמל ||. שני אופרטורים אלה נקראים **אופרטורים לוגים מתנים** (conditional logical operators) תפקידם לשלב בין ביטויים בוליאניים כדי ליצור ביטויים בוליאניים מורכבים יותר. אופרטורים בינאריים אלה דומים לאופרטורים של השוואה ויחס בכך שהם מחזירים ערך **true** או **false**, אולם הם שונים מהם בכך שהערכים עליהם הם מופעלים חייבים להיות גם הם **true** או **false**.

תוצאת האופרטור && תהיה **true** אך ורק במקרה ששני הביטויים הבוליאניים שהוא מופעל עליהם יהיו גם **true**. לדוגמה, המשפט הבא יציב את הערך אמת במשתנה **validPercentage** (ערך אחוז חוקי), רק אם ערכו של **percent** גדול או שווה לאפס **וגם** קטן או שווה למאה:

```
bool validPercentage;
validPercentage = (percent >= 0) && (percent <= 100);
```

טיפ

טעות נפוצה בקרב מתכנתים מתחילים היא לנסות ולשלב את שתי הבדיקות מבלי להקליד פעמיים את שם המשתנה. הנה כך:

```
percent >= 0 && <= 100; // this statement will not compile
```

שימוש בסוגריים עוזר להימנע מסוג זה של טעויות, וגם עוזר להבהיר את מטרת הביטוי. השווה לדוגמה בין שני הביטויים הבאים:

```
validPercentage = percent >= 0 && percent <= 100;
```

לעומת

```
validPercentage = (percent >= 0) && (percent <= 100);
```

שני הביטויים מחזירים את אותו הערך, כי הקדימות (precedence) של האופרטור && נמוכה מזו של האופרטורים =< ו-=>. אולם הביטוי השני ברור ומובן יותר.

תוצאת האופרטור || תהיה אמת, אם הערך של אחד מהביטויים הבוליאניים שהוא מופעל עליהם הוא **true**. האופרטור || מיועד לבדוק אם קיים ביטוי שערכו **true** בסדרה של ביטויים בוליאניים. לדוגמה, המשפט הבא יציב את הערך **true** במשתנה **invalidPercentage** (ערך אחוז לא חוקי) אם ערכו של **percent** קטן מאפס **או** גדול ממאה:

```
bool invalidPercentage;
invalidPercentage = (percent >= 0) || (percent <= 100);
```

קיצור דרך

לאופרטורים && ו-|| יש מאפיין בשם **קיצור דרך** (**short circuiting**). לעיתים אין צורך לבדוק את שני האופרנדים. לדוגמה, אם האופרנד השמאלי של אופרטור && הוא **false**, משמעות הדבר היא שתוצאת הביטוי כולו תהיה גם היא **false**, ללא חשיבות מה יהיה ערכו של האופרנד הימני. כמו כן, אם האופרנד השמאלי של אופרטור || הוא **true**, אז תוצאת הביטוי כולו תהיה תמיד **true**. במקרים כאלה, האופרטורים && ו-|| מדלגים על בדיקת הביטוי הבוליאני הימני. להלן מספר דוגמאות:

```
(percent >= 0) && (percent <= 100)
```

בביטוי זה, אם הערך של **percent** קטן מאפס, הביטוי הבוליאני בצידו השמאלי של && יהיה **false**. משמעות הדבר היא שהביטוי כולו חייב להחזיר ערך **false**, יהיה ערכו של הביטוי הימני אשר יהיה. לפיכך, האופרטור && לא יבדוק את הביטוי הבוליאני שמימינו.

```
(percent < 0) || (percent > 100)
```

בביטוי זה, אם הערך של **percent** קטן מאפס, הרי שהביטוי הבוליאני בצידו השמאלי של || יהיה **true**. משמעות הדבר היא שהביטוי כולו חייב להניב ערך **true**, יהיה ערכו של הביטוי הימני אשר יהיה. לפיכך, האופרטור || לא יבצע בדיקה של הביטוי הבוליאני שמימינו.

תכנון זהיר של ביטויים הכוללים אופרטורים לוגיים מתנים יכול לשפר את ביצועי הקוד בכך שתמנע פעולות מיותרות. ביטויים בוליאניים פשוטים וקלים לבדיקה עליך למקם מצידו השמאלי של האופרטור הלוגי המתנה, בעוד שאת הביטויים המורכבים יותר עליך למקם מצידו הימני. במקרים רבים תראה שהתוכנית לא תצטרך לבדוק את הביטויים המורכבים כלל.

סיכום הקדימות והשיוך של האופרטורים השונים

הטבלה הבאה מסכמת את הקדימות (precedence) והשיוך (associativity) של כל האופרטורים שלמדת עד כה. אופרטורים המשתייכים לאותה קטגוריה הם בעלי קדימות זהה ומבוצעים לפי סדרם משמאל לימין. הקטגוריות מסודרות בסדר יורד לפי רמת הקדימות שלהן (הקטגוריה הראשונה היא בעלת הקדימות הגבוהה ביותר).

Category	Operators	Description	Associativity
Primary	( )	Precedence override	Left
	++	Post-increment	
	- -	Post-decrement	
Unary	!	Logical NOT	Left
	+	Addition	
	-	Subtraction	
	++	Pre-increment	
	- -	Pre-decrement	

Category	Operators	Description	Associativity
Multiplicative	* / %	Multiply Divide Division remainder	Left
Additive	+ -	Addition Subtraction	Left
Relational	< <= > >=	Less than Less than or equal Greater than Greater than or equl	Left
Equality	== !=	Equal to Not equal to	Left
Conditional AND	&&	Logical AND	Left
Conditional OR	||	Logical OR	Left
Assignment	=		Right

שימוש במשפטי if לקבלת החלטות

עליך להשתמש במשפטי **if** כאשר ברצונך להחליט על הפעלה של אחד משני בלוקים של קוד על פי ערך של ביטוי בוליאני.

התחביר של משפטי if

התחביר של משפטי **if** נראה כך (**if** ו-**else** הן מילות מפתח):

```
if( booleanExpression )
   statement-1;
else
   statement-2;
```

אם הערך של **booleanExpression** הוא **true**, יופעל **statement-1**; אחרת, הערך של **booleanExpression** חייב להיות **false**, וכתוצאה מכך יופעל **statement-2**. בסיום, התוכנית תעבור אל הביטוי שאחרי הפקודה if. מילת המפתח else והמשפט שאחריה הינם תוספת רשות למשפט **if**. אם אין סעיף **else**, לא יבוצע דבר אם הערך של **booleanExpression** הוא **false**, והתוכנית תעבור לביטוי הבא.

לדוגמה, להלן משפט **if** שמוסיף 1 לשדה השניות בשעון עצר (הדקות אינן רלוונטיות כעת). אם הערך של **seconds** הוא 59, הקוד יציב בו 0, אחרת (else) הוא יוסיף לו 1 באמצעות האופרטור ++:

```
int seconds;
...
if (seconds == 59)
    seconds = 0;
else
    seconds++;
```

רק ביטויים בוליאניים בבקשה!

הביטוי שבמשפט **if** חייב להיות בין צמד סוגריים, וחייב להיות ביטוי בוליאני. במספר שפות אחרות (C ו-C++ למשל), ניתן לכתוב ביטוי המחזיר מספר שלם והמהדר ימיר את הערך השלם ל-**true** (לערך שאינו אפס) או ל-**false** (לערך אפס). שפת C# אינה תומכת באפשרות הזו, וניסיון להציב ביטוי שאינו בוליאני במשפט if יגרום לשגיאה בשלב ההידור.

אם תחליף בטעות את אופרטור ההשוואה (==) באופרטור ההצבה (=) במשפט **if**, המהדר יזהה את הטעות שלך. לדוגמה:

```
int seconds;
...
if(seconds = 59) // compile-time error
...
if(seconds == 59) // ok
```

הצבות שגויות הן מקור שכיח נוסף לבאגים בשפות C ו-C++. הן ממירות באופן אוטומטי את הערך המוצב (59, במקרה זה) לביטוי בוליאני ללא ידיעת המשתמש. למעשה, כל ערך שאינו אפס נחשב לאמת. כתוצאה, הקוד שמצורף למשפט **if** היה מופעל תמיד.

ניתן גם להשתמש במשתנה בוליאני בתור הביטוי, כמודגם להלן:

```
bool inWord = false;
...
if (inWord == true)  // ok, but not commonly used
...
if(inWord)           // better
```

שימוש בבלוקים לקיבוץ משפטים

לעיתים צריך להפעיל שני משפטים או יותר אם וכאשר הביטוי הבוליאני מתגלה כאמת. דרך אחת לעשות זאת תהיה לקבץ את המשפטים בתוך שיטה ולקרוא לה. דרך פשוטה יותר תהיה לקבץ את המשפטים **בבלוק** (**block**), שאינו אלא רצף של משפטים המתוחמים בין צמד סוגריים מסולסלים. בדוגמה הבאה, שני המשפטים המאתחלים את המשתנה **seconds** ומוסיפים 1 לערך המשתנה **minutes** מקובצים יחדיו בבלוק, והבלוק כולו מופעל כאשר ערך **seconds** שווה 59:

```
int seconds = 0;
int minutes = 0;
...
if(seconds == 59)
{
    seconds = 0;
    minutes++;
}
else
    seconds++;
```

חשוב

אם תשמיט את הסוגריים המסולסלים, המהדר של #C ישייך למשפט **if** את המשפט הראשון (seconds = 0;) בלבד. בזמן הידור התוכנית, המהדר לא יראה במשפט השני (minutes++;) חלק ממשפט **if**. בנוסף לכך, כאשר המהדר מגיע למילת המפתח **else**, הוא לא יקשר אותה למשפט if, ואז ידווח על שגיאת הידור.

קינון משפטי if

ניתן להציב משפטי **if** בתוך משפטי **if** אחרים, בכמה רמות. בצורה זו ניתן לשרשר רצף של ביטויים בוליאניים הנבדקים זה אחר זה עד אשר אחד מהם מתגלה כ-**true**. בדוגמה הבאה, הערך של **day** הוא 0, ולכן הבדיקה הראשונה מחזירה ערך **true** והערך **Sunday** מוצב במשתנה **dayName**. אם הערך של **day** אינו 0, הבדיקה הראשונה תיכשל והשליטה תעבור לסעיף **else**, אשר יפעיל את משפט **if** השני שבו הערך של **day** מושווה ל-1. המשפט **if** השני מופעל רק אם המשפט הראשון הוא **false**. המשפט השלישי מופעל רק אם המשפט הראשון וגם המשפט השני הם **false**, וכן הלאה עד לסיום שמציג "לא ידוע" כאשר הערך אינו מוכר.

```
if (day == 0)
    dayName = “Sunday”;
else if(day == 1)
    dayName = “Monday”;
else if(day == 2)
    dayName = “Tuesday”;
else if(day == 3)
    dayName = “Wednesday”;
else if(day == 4)
    dayName = “Thursday”;
else if(day == 5)
    dayName = “Friday”;
else if(day == 6)
    dayName = “Saturday”;
else
    dayName = “unknown”;
```

בתרגיל הבא עליך לכתוב שיטה המשתמשת במשפט **if** מדורג כדי להשוות בין שני תאריכים.

כתיבת משפטי if

1. הפעל את Microsoft Visual Studio 2005.
2. פתח את הפרויקט Selection שבתיקייה My Documents\Microsoft Press\Visual Csharp Step by Step\Chapter 4\Selection.
3. בתפריט Debug בחר Start Without Debugging.

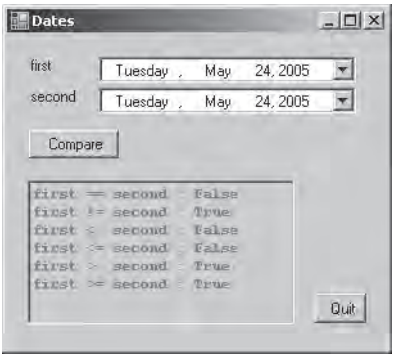
סביבת הפיתוח תבנה ותפעיל את היישום Windows. בטופס יש שני פקדים מסוג **DateTimePicker** שנקראים **first** ו-**second**, אשר מציגים לוח שנה המאפשר לבחור מתוכו תאריך. שני הפקדים מציגים כרגע את התאריך הנוכחי.

4. לחץ Compare.

המלל הבא יופיע בתיבת הטקסט:

```
first == second : False
first != second : True
first <  second : False
first <= second : False
first >  second : True
first >= second : True
```

הביטוי הבוליאני first == second אמור להחזיר true (אמת), מכיוון שגם **first** וגם **second** מקובעים על התאריך הנוכחי. למעשה, האופרטורים היחידים המציגים תוצאה נכונה הם "קטן מ-" ו"גדול או שווה ל-".



5. לחץ Quit.

כתוצאה, תחזור לסביבת הפיתוח של Visual Studio 2005.

6. הצג את הקוד עבור Form1.cs בחלון Code and Text Editor. אתר את השיטה **compare_Click**, אשר נראית כך:

```
private void compare_Click(object sender, System.EventArgs e)
{
    int diff = dateCompare(first.Value, second.Value);
    info.Text = "";
    show("first == second", diff == 0);
    show("first != second", diff != 0);
    show("first <  second", diff < 0);
    show("first <= second", diff <= 0);
    show("first >  second", diff > 0);
    show("first >= second", diff >= 0);
}
```

השיטה מופעלת בכל פעם שהמשתמש לוחץ על הלחצן **Compare** שבטופס. היא משווה בין ערכי התאריכים המוצגים בפקדים **first** ו-**second** באמצעות שיטה אחרת בשם **dateCompare**. בשלב הבא תבחן את פעולת השיטה **dateCompare**, שמטרתה לבדוק את הארגומנטים שהיא מקבלת ולהחזיר ערך שלם המבוסס על היחס ביניהם. היא מחזירה 0 כאשר הם בעלי ערך שווה, 1- כשערך **first** קטן מערך **second**, ומחזירה 1+ כשערך **first** גדול מערך **second**. זכור שבהשוואת תאריכים, תאריך מסוים נחשב גדול יותר מתאריך אחר אם הוא מאוחר יותר מבחינה כרונולוגית.

השיטה **show** מציגה את תוצאות ההשוואה בתיבת הטקסט info שבטופס.

7. אתר את השיטה **dateCompare** הנראית כך:

```
private int dateCompare(DateTime leftHandSide, DateTime
rightHandSide)
{
    // TO DO
    return 42;
}
```

נכון לעכשיו, שיטה זו מחזירה תמיד את אותו הערך, ולא 0, 1- או 1+ בהתאם לערכי הפרמטרים שלה. כעת ברור מדוע היישום אינו פועל כשורה. עליך לכתוב את השיטה בצורה נכונה, כדי שתשווה בין שני התאריכים.

8. הסר את ההערה TO DO // ואת משפט ההחזרה (**return**) מהשיטה **dateCompare**.

9. הקלד את המשפטים הבאים בגוף השיטה **dateCompare**.

```
int result;
if (leftHandSide.Year < rightHandSide.Year)
    result = -1;
else if (leftHandSide.Year > rightHandSide.Year)
    result = +1;
else if (leftHandSide.Month < rightHandSide.Month)
    result = -1;
else if (leftHandSide.Month > rightHandSide.Month)
    result = +1;
else if (leftHandSide.Day < rightHandSide.Day)
    result = -1;
else if (leftHandSide.Day > rightHandSide.Day)
    result = +1;
else
    result = 0;
return result;
```

אם הביטוי leftHandSide.Year < rightHandSide.Year וגם הביטוי leftHandSide.Year > rightHandSide.Year הם שקר, אז הביטוי leftHandSide.Year == rightHandSide.Year חייב להיות אמת. התוכנית ממשיכה אל ההשוואה בין המאפיין Month של lhs לבין המאפיין Month של rhs.

כמו קודם, אם הביטוי leftHandSide.Month < rightHandSide.Month וגם הביטוי leftHandSide.Month > rightHandSide.Month הם שקר, אז הביטוי leftHandSide.Month == rightHandSide.Month חייב להיות אמת, והתוכנית תמשיך שוב,

הפעם להשוואה בין המאפיין **Day** של **lhs** לבין **Day** של **rhs**.

גם במקרה זה, אם הביטוי leftHandSide.Day < rightHandSide.Day וגם הביטוי leftHandSide.Day > rightHandSide.Day הם שקר, אז הביטוי leftHandSide.Day == rightHandSide.Day חייב להיות אמת. מכיוון שהמאפיינים **Year** ו-**Month** גם הם בהכרח אמת, שני התאריכים הם ללא ספק שווים.

10. בתפריט Debug בחר Start Without Debugging (הפעלה ללא ניפוי). היישום ייבנה ויופעל מחדש. גם הפעם שני הפקדים מציגים את התאריך הנוכחי.

11. לחץ Compare.

המלל הבא יופיע בתיבת הטקסט:

```
first == second : True
first != second : False
first <  second : False
first <= second : True
first >  second : False
first >= second : True
```

אלו הן התוצאות הנכונות.

12. שנה את התאריך השני כדי שיהיה התאריך של מחר.

13. לחץ Compare.

המלל הבא יופיע בתיבת הטקסט:

```
first == second : False
first != second : True
first <  second : True
first <= second : True
first >  second : False
first >= second : False
```

גם הפעם תופענה התוצאות הנכונות.

14. לחץ Quit.

אופרטור טרנארי

אופרטור טרנארי (Ternary) הינו אופרטור התניה אשר מחזיר אחד משני ערכים, בהתבסס על ערך של תנאי בוליאני. התחביר נראה כך:

```
condition ? firstTrueExpression : secondFalseExpression;
```

החלק הראשון בביטוי, כלומר condition, הוא תנאי (בוליאני); לאחריו נכתב **?** ולאחר מכן שני ערכים מופרדים בנקודתיים (:).

אם התנאי מחזיר אמת, יוחזר הערך הראשון; אחרת, במצב שקר, יוחזר הערך השני.

אופרטור זה הינו גרסה מקוצרת למשפט **if**. רצוי להימנע משימוש באופרטור זה אלא להשתמש במשפט if, מכיוון שמשפטי if קלים יותר לקריאה ולהבנה.

בדוגמה הבאה, אם ציון המבחן קטן משישים, מוחזר הערך Fail; אחרת, כאשר הציון גדול משישים, מוחזר הערך Pass.

```
testStatus = (testScore < 60) ? “Fail” : “Pass”;
```

הביטוי שלעיל שקול למשפט **if** הבא:

```
if (testScore < 60)
   testStatus = “Fail”;
else
   testStatus = “Pass”;
```

משפטי Switch

לעיתים, משפטי **if** מקוננים דומים מאוד זה לזה, כי כולם בודקים את אותו הביטוי. ההבדל היחיד הוא בכך שכל אחד מהם משווה את תוצאת הביטוי לערך אחר. לדוגמה:

```
if (day == 0)
    dayName = “Sunday”;
else if(day == 1)
    dayName = “Monday”;
else if(day == 2)
    dayName = “Tuesday”;
else if(day == 3)
 ...
else
    dayName = “unknown”;
```

במצבים כאלה אפשר לכתוב את המשפט המקונן כמשפט switch (מתג), כדי לגרום לכך שהתוכנית תהיה יותר קריאה וקלה להבנה.

הבנת התחביר של משפטי switch

התחביר של משפט **המתג switch** כולל כמה מילות מפתח: **switch**, **case** ו-**default**:

```
switch ( controllingExpression )
{
    case constantExpression :
        statements
        break;
    case constantExpression :
        statements
        break;
    ...
    default :
        statements
        break;
}
```

בתחביר של משפט **switch** מופיע הערך **controllingExpression**, אשר יכול להיות תוצאה של ביטוי, או משתנה. ערך זה מושווה אחר כך לכל אחד מהערכים בכל אחד ממשפטי **case** (**constantExpression**) עד שנמצא ערך תואם. אם לא נמצא ערך תואם, הזרימה עוברת אל ה-**case** של ברירת המחדל (**default**).

הערה

אם ערך הביטוי **controllingExpression** אינו שווה לאחד מהמקרים (**case**), ואין במשפט **switch** גם משפט **default**, התוכנית מפעילה את המשפט הראשון שאחרי הסוגר המסולסל הימני של המשפט **switch**.

לדוגמה, את משפט **if** המקונן היה ניתן לנסח גם בתור משפט **switch** כזה:

```
private void copyOne(char current)
{
    switch (day)
    {
    case 0 :
        dayName = “Sunday”;
        break;
    case 1 :
        dayName = “Monday”;
        break;
    case 2 :
        dayName = “Tuesday”;
        break;
    ...
    default :
        dayName = “unknown”;
        break;
    }
```

שמירה על חוקי המשפט switch

משפטי **switch** שימושים ביותר, אך למרבה הצער לא ניתן להשתמש בהם תמיד. כל משפט **switch** חייב לקיים את החוקים הבאים:

- ניתן להשתמש במשפטי **switch** רק עבור סוגי נתונים פרימיטיביים, כגון **int** ו-**string**. עבור סוגי נתונים אחרים צריך להשתמש במשפטי **if**.
- התוויות של המקרים (**case**) השונים חייבות להיות קבועות, למשל 42 או "42". אם עליך לחשב את התווית של כל מקרה בזמן ההרצה, צריך להשתמש במשפט **if**.
- התוויות של המקרים השונים המוצגים במשפט חייבות להיות ביטויים ייחודיים. במילים אחרות, לא ייתכן שלשתי תוויות יהיה אותו הערך.
- ניתן להפעיל את אותם המשפטים עבור יותר מערך אחד, באמצעות כתיבת מספר **case** labels ברצף, מבלי להוסיף להם משפטים. במקרה כזה, הקוד שאחרי התווית האחרונה ברצף יופעל עבור כל אחד מהמקרים. אולם אם אחת התוויות הללו מכילה משפט אחד או יותר, הפעלת הקוד לא תצליח והמהדר יודיע על שגיאה. לדוגמה:

```
switch (trumps)
{
case Hearts :    // Fall-through allowed - no code between labels
case Diamonds :     // code execued for Hearts and Diamonds
   color = “RED”;
   break;
case Clubs :
   color = “Black”;
   break;
case Spades :       // Error - code between labels
   color = “Black”;
   break;
}
```

הערה

משפט **שבירה** (**break**) היא הדרך הנפוצה ביותר לעצור את ההתקדמות (fall-through), אולם ניתן להשתמש גם במשפט החזרה (**return**) או משפט זריקה (**throw**). בפרק 6 ניתן הסבר על משפטי הזריקה.

הפסקת התקדמות

הכלל של הפסקת ההתקדמות (no fall-through rule) מונע שינוי של סדר הפעולות במשפט **switch**, מבלי לשנות את משמעות המשפט (כולל את התווית **default** אשר נהוג להציבה בסוף, גם אם אין חובה לעשות זאת).

מתכנתים מרקע תכנות בשפות C ו- C++ צריכים לשים לב לכך שחובה להציב משפט שבירה בסוף כל מקרה במשפט switch, ואפילו עבור מקרה ברירת המחדל. דרישה זו היא יתרון, כי מתכנתים אלה שוכחים לעיתים קרובות את משפטי השבירה, וכך מאפשרים להתקדם לתווית הבאה ולגרום לבאגים אשר קשים מאוד לאבחון.

ניתן לביים את התקדמות ביצוע התוכנית ב-C# באמצעות משפט **דילוג goto**, כדי לדלג אל המקרה הבא או אל ברירת המחדל. לא מומלץ לנהוג כך, ולכן גם אין הסבר בספר זה כיצד לעשות זאת!

בתרגיל הבא עליך להשלים תוכנית הקוראת את התווים של מחרוזת מסוימת וממפה את הייצוג XML של כל תו. לדוגמה, לתו ">" יש משמעות מיוחדת ב-XML (משמש ליצירת אלמנטים) ויש לתרגמו ל-"<". עליך לכתוב משפט **switch** אשר בוחן את ערך התו, ולוכד תווים מיוחדים של XML באמצעות תוויות מקרים.

כתוב משפטי switch

1. הפעל את Visual Studio 2005.
2. פתח את הפרויקט SwitchStatement שבתיקייה My Documents\Microsoft Press\Visual Csharp Step by Step\Chapter 4\SwitchStatment.
3. בתפריט Debug בחר Start Without Debugging.

סביבת הפיתוח תבנה ותפעיל את היישום. בטופס יש שתי תיבות טקסט המופרדות על ידי לחצן Copy.

4. הקלד את הטקסט הבא בתיבת הטקסט העליונה:

```
inRange = (lo <=number) && (number <= lo);
```

5. לחץ Copy.

המשפט יועתק במדויק אל תיבת הטקסט התחתונה, ללא תרגום של התו "<".

6. סגור את הטופס.

7. הצג את הקוד עבור Form1.cs בחלון Code and Text Editor. אתר את השיטה **copyOne**.

השיטה **copyOne** מעתיקה תו אחד מתיבת הטקסט העליונה אל התחתונה. השיטה מכילה כעת משפט **switch** אחד בעל ברירת מחדל בלבד.

במספר השלבים הבאים, תשנה את השיטה כדי שתמיר תווים בעלי חשיבות ב-XML לצורת המיפוי שלהם ב-XML. לדוגמה, התו "<" יומר למחרוזת "<".

8. הוסף את המשפטים הבאים אל משפט **switch**, לפני התווית **default**.

```
case '<':
    target.Text += "&lt;";
    break;
case '>':
    target.Text += "&gt;";
    break;
case '&':
    target.Text += "&amp;";
    break;
case '\"':
    target.Text += "&#34;";
    break;
case '\'':
    target.Text += "&#39;";
    break;
```

הערה

הלוכסן ההפוך (\) שנראה בשני המקרים האחרונים הוא תו Escape הגורם לתוכנית להתייחס לתווים שאחריו (" ו- ") במובן מילולי, ולא בתור תווים לסימון מחרוזת או תווים רגילים.

9. בתפריט Debug בחר Start Without Debugging.

סביבת הפיתוח תבנה ותפעיל את היישום.

10. הקלד את המשפט הבא בתיבת הטקסט העליונה:

```
inRange = (lo <=number) && (number <= lo);
```

11. לחץ Copy.

הקוד יועתק לתיבת הטקסט התחתונה. הפעם, כל תו עובר מיפוי XML על ידי המשפט switch.

12. סגור את הטופס.

☯ **אם ברצונך להמשיך לפרק הבא**

השאר את Visual Studio 2005 פעילה ועבור לפרק 5.

☯ **אם ברצונך לסגור את Visual Studio 2005**

בתפריט File בחר Exit. אם מופיעה תיבת דו-שיח Save, לחץ Yes.

פרק 4 – טבלה מסכמת

המשימה	צריך	דוגמה
לקבוע אם שני ערכים שווים זה לזה.	להשתמש באופרטור == או =!.	answer = = 42
להשוות ערכים של שני ביטויים.	להשתמש באופרטור <, =>, > או =<.	age >= 21
להכריז על משתנה בוליאני.	להשתמש במילת המפתח bool בתור סוג המשתנה.	bool inRange;
ליצור ביטוי בוליאני המחזיר ערך true (אמת), רק אם שני תנאים אחרים הם אמת.	להשתמש באופרטור &&.	inRange = (lo <= number) && (number <= hi);
ליצור ביטוי בוליאני המחזיר ערך אמת אם לפחות אחד משני תנאים אחרים הוא אמת.	להשתמש באופרטור ||.	inRange = (number < lo) || (hi < number);
להפעיל משפט אם תנאי כלשהו מתקיים.	להשתמש במשפט if.	If (inRange) Process();

להפעיל יותר ממשפט אחד אם התנאי מתקיים.	להשתמש בבלוק (block).	If (seconds = = 59)
		{
		seconds = 0;
		minutes++;
		}

ליצור שיוך בין משפטים שונים לבין ערכים שונים של ביטוי שליטה (controlling expression).	להשתמש במשפט switch.	Switch (current)
		{
		Case '<' :
		…
		break;
		default :
		...
		break;
		}

פרק 5

הצבה מורכבת ומשפטי איטרציה

בסיום פרק זה, תוכל:

- לעדכן את ערך המשנה באמצעות אופרטורים להצבה מורכבת.
- לכתוב משפטי איטרציה מסוג **while**, **for** ו-**do**.
- לסקור שיטה מסוג **do**, ולראות כיצד הערכים משתנים.

בפרק 4 למדת להשתמש במבנים **if** ו-**switch** כדי להפעיל משפטים באופן סלקטיבי. בפרק זה תלמד להשתמש במגוון משפטי איטרציה (כלומר iteration או **looping**), כדי להפעיל שוב ושוב משפט אחד או יותר. בעת כתיבת משפטי איטרציה, עליך לשלוט בדרך כלל במספר החזרות שיבוצעו. תוכל לעשות זאת באמצעות משתנה שערכו יעודכן בכל חזרה. התהליך ייעצר כאשר ערך המשתנה יגיע לערך מסוים, ולכן עליך ללמוד גם על האופרטורים המיוחדים להצבה, שתשתמש בהם כדי לעדכן את ערך המשתנה בנסיבות אלו.

אופרטורים להצבה מורכבת

למדת כבר כיצד להשתמש באופרטורים כדי ליצור ערכים חדשים. לדוגמה, המשפט הבא משתמש באופרטור "+" כדי ליצור ערך גדול ב-42 מהמשתנה **answer** ולהציג אותו במסך.

```
Console.WriteLine(answer + 42);
```

למדת גם כיצד להשתמש במשפטי הצבה כדי לשנות את ערך המשתנה. המשפט הבא משתמש באופרטור ההצבה, כדי לשנות את ערך המשתנה **answer** ל-42:

```
answer = 42;
```

אם ברצונך להוסיף לערך הנוכחי של משתנה, עליך לשלב את אופרטור החיבור ואופרטור ההצבה. לדוגמה, המשפט הבא מוסיף 42 למשתנה **answer**. במילים אחרות, לאחר הפעלת המשפט, הערך של **answer** יהיה גדול ב-42 מערכו הקודם:

```
answer = answer + 42;
```

למרות שמשפט זה יבצע את המשימה, מתכנתים מקצועיים לא ישתמשו בו. הוספת ערך למשתנה היא פעולה שכיחה מאוד, ולכן השפה #C מאפשרת לבצע אותה בצורה מקוצרת באמצעות האופרטור "+=" המשמש להצבה מורכבת. מתכנת מקצועי יוסיף 42 למשתנה **answer** על ידי פקודת הקיצור הזו:

```
answer += 42;
```

ניתן להשתמש בקיצור זה לחיבור של כל משתנה אריתמטי עם משתנה ההצבה, כפי שמוצג בטבלה הבאה. קבוצת האופרטורים הזו מכונה **אופרטורים להצבה מורכבת** (**compound assignment operators**).

Don't write this	Write this
variable = variable * number;	variable *= number;
variable = variable / number;	variable /= number;
variable = variable % number;	variable %= number;
variable = variable + number;	variable += number;
variable = variable - number;	variable -= number;

טיפ

הקדימות והשיוך הימני של אופרטורים להצבה מורכבת זהים לאלה של אופרטור ההצבה.

ניתן ליישם את האופרטור += גם על מחרוזות, כדי להוסיף מחרוזת אחת בסופה של אחרת. לדוגמה, הקוד הבא יגרום להדפסת המחרוזת "Hello John" במסך:

```
string name = "John";
string greeting = "Hello ";
greeting += name;
Console.WriteLine(greeting);
```

את האופרטורים האחרים בקבוצה זו לא ניתן ליישם על מחרוזות.

הערה

כאשר ברצונך להגדיל או להקטין את הערך באחד, השתמש באופרטורים "++" ו- "- -" (מינוס, כפול וצמוד), ולא באופרטורים להצבה מורכבת. לדוגמה, החלף את:

```
count += 1;
```

עם:

```
count++;
```

כתיבת משפטים מסוג While

משפטים מסוג **while** (**כל עוד**) משמשים לחזרה על משפט כלשהו כל עוד ביטוי בוליאני כלשהו הוא אמת (true). התחביר של משפט **while** הוא:

```
while ( booleanExpression )
    statement;
```

הביטוי הבוליאני נבדק ואם הוא אמת, המשפט יופעל והביטוי ייבדק שוב. אם הביטוי עודנו אמת, המשפט יופעל שוב והביטוי ייבדק שוב. התהליך יימשך עד אשר הביטוי הבוליאני יהיה שקר (false) ואז משפט **while** יסתיים.

למשפט מסוג **while** יש מספר קווי דמיון סמנטיים עם משפט מסוג **if** (ולמעשה התחביר של שניהם זהה, לבד ממילת המפתח):

- הביטוי חייב להיות ביטוי בוליאני.
- הביטוי הבוליאני חייב להיות בין סוגריים.
- אם הביטוי הבוליאני הוא שקר בבדיקה הראשונה, המשפט לא יופעל כלל.
- אם ברצונך להפעיל שני משפטים או יותר תחת שליטת משפט **while**, עליך להשתמש בסוגריים מסולסלים כדי לתחום משפטים אלו בבלוק אחד.

להלן משפט **while** אשר מציג במסך את הערכים השלמים מ-0 עד 9:

```
int i = 0;
while (i != 10)
{
    Console.WriteLine(i);
    i++;
}
```

כל המשפטים מסוג **while** חייבים להיפסק בשלב מסוים. טעות שכיחה של מתחילים היא לשכוח הוספת משפט שיגרום לביטוי הבוליאני להפוך בסופו של דבר לשקר, וכך להפסיק את הלולאה (loop). בדוגמה שלעיל עושה זאת המשפט **i++;**.

הערה

המשתנה (i) בלולאה **while** שולט במספר החזרות אשר יבוצעו. זהו מבנה נפוץ מאוד, והמשתנה הממלא את התפקיד הזה נקרא לעיתים קרובות משתנה **זקיף** (**Sentinel**).

בתרגיל הבא, עליך לכתוב לולאת **while** שקוראת את תוכן קובץ המקור שורה אחר שורה, וכותבת כל שורה בתיבת טקסט ביישום של Windows.

כתוב משפט while

1. באמצעות סביבת הפיתוח Visual Studio 2005, פתח את הפרויקט **WhileStatements**, שבתיקייה My Documents\Microsoft Press\Visual CSharp Step by Step\Chapter 5\ WhileStatement.

2. בתפריט Debug בחר Start Without Debugging.

סביבת הפיתוח תבנה ותפעיל את היישום Windows. היישום עצמו הוא תוכנה פשוטה לצפייה בקבצי טקסט, אשר מאפשרת לבחור בקובץ ולהציג את תוכנו.

3. לחץ Open File.

על המסך תופיע תיבת הדו-שיח Open.

4. פתח את התיקייה My Documents\Microsoft Press\Visual CSharp Step by Step\Chapter 5\WhileStatement\WhileStatement.

5. בחר בקובץ Form1.cs ולחץ Open.

שם קובץ המקור (Form1.cs) יופיע בתיבת הטקסט הקטנה, אולם התוכן של הקובץ אינו מופיע בתיבת הטקסט הגדולה. הסיבה היא שטרם נכתב הקוד אשר קורא את קובץ המקור ומציג את תוכנו בתיבת הטקסט הגדולה. בצעדים הבאים תוסיף את הרכיב הזה לתוכנית.

6. סגור את הטופס וחזור לסביבת הפיתוח.

7. הצג את הקוד עבור הקובץ Form1.cs בחלון Code and Text Editor. אתר את השיטה **openFileDialog_FileOk**.

שיטה זו מופעלת כאשר המשתמש לוחץ Open לאחר בחירת הקובץ בתיבת הדו-שיח Open. בשלב זה, גוף השיטה נראה כך:

```
string fullPathname = openFileDialog.FileName;
FileInfo src = new FileInfo(fullPathname);
filename.Text = src.Name;
/* add while loop here */
```

המשפט הראשון מכריז על משתנה **fullPathname** מסוג **string** ומאתחל אותו לערך שבמאפיין **FileName** של האובייקט **openFileDialog**. משפט זה מאתחל את **fullPathname** לשם המלא (כולל התיקייה) של קובץ המקור שנבחר בתיבה Open.

הערה

האובייקט **openFileDialog** הוא מופע (instance) של המחלקה **OpenFileDialog** שנמצאת בערכת הכלים. מחלקה זו כוללת שיטות שבעזרתן ניתן להציג את תיבת הדו-שיח Open הסטנדרטית של Windows, לבחור קובץ ולשלוף את השם או הנתיב של קובץ נבחר.

המשפט השני מכריז על משתנה src מסוג **FileInfo** ומאתחל אותו לאובייקט המייצג את הקובץ שנבחר בתיבת הדו-שיח Open (**FileInfo** היא מחלקה של Microsoft .Net Framework המאפשרת לערוך קבצים).

המשפט השלישי מבצע השמה של המאפיין **Text** מהפקד **filename** מסוג תיבת טקסט, אל המאפיין **Name** של המשתנה **src**. המאפיין name במשתנה src מכיל את השם המלא של הקובץ שנבחר בתיבת הדו-שיח Open, אך לא מציין את התיקייה. ההשמה גורמת לשם הקובץ להופיע בתיבת הטקסט filename שבטופס.

8. החלף את הערה **/ *add while loop here* /** להוספת הלולאה, במשפט הבא:

```
source.Text = “”;
```

השדה **source** הוא תיבת טקסט גדולה בטופס. הצבת המחרוזת הריקה (“”) במאפיין **Text** תגרום למחיקת הטקסט המוצג כרגע.

9. הקלד את המשפט הבא לאחר השורה שהוספת עכשיו לשיטה **openFileDialog_FileOk**:

```
TextReader reader = src.OpenText();
```

משפט זה מכריז על משתנה מסוג **TextReader** בשם **reader** (**TextReader** הינה מחלקה נוספת של NET Framwork., המאפשרת לקרוא רצפי תווים מתוך קבצים, למשל. היא נמצאת במרחב השמות System.IO). המחלקה **FileInfo** מספקת את השיטה **OpenText** המשמשת לפתיחת קבצים שנבחרו על ידי המשתמש בתיבת הדו-שיח Open. השיטה **OpenText** מחזירה אובייקט **TextReader**. משפט זה מאתחל את המשתנה **reader** לאובייקט **TextReader**, אשר הוחזר מהקריאה לשיטה **src.OpenText**. המשתנה **reader** יכול לשמש כעת לקריאת הקובץ שנבחר על ידי המשתמש.

10. הקלד את המשפטים הבאים אחרי השורה הקודמת, אשר הוספת לשיטה **openFileDialog_FileOk**:

```
string line = reader.ReadLine();
while (line != null)
{
    source.Text += line + “\n”;
    line = reader.ReadLine();
}
reader.Close();
```

הקוד מכריז על משתנה **line** מסוג **string**, המשמש לאחסן שורת טקסט בשעה שהמשתמש קורא אותה מהקובץ. המשפט קורא לשיטה **reader.ReadLine** כדי שתקרא את השורה הראשונה מהקובץ. שיטה זו מחזירה את שורת הטקסט הבאה או לחילופין, ערך מיוחד הנקרא **null** כאשר אין יותר שורות. התוצאה המוחזרת מהקריאה מוצבת במשתנה **line**.

הביטוי הבוליאני בתחילת הלולאה מסוג **while** בודק את הערך במשתנה line. אם אינו null, גוף הלולאה יציג את שורת הטקסט על ידי סיפוחה לסוף המאפיין **Text** בתיבת הטקסט **source**, לצד תו מעבר שורה (”\n“ ־ השיטה **ReadLine** של האובייקט **TextReader** מורידה את כל מעברי השורות בכל שורה חדשה, ולכן יש להחזיר אותם כחלק מהקוד). הלולאה קוראת את שורת הטקסט הבאה (זהו החלק בלולאה שאחראי לעדכון) לפני חזרה נוספת.

כאשר הלולאה מסתיימת, הקריאה לשיטה **Close** של האובייקט **TextReader** סוגרת את הקובץ.

טיפ

כשתהיה מתורגל יותר בתחביר של C#, תוכל לשנות את הקוד של לולאת while באופן הבא:

```
string line;
while ((line = reader.ReadLine()) != null) { source.Text += line + '\n'; }
reader.Close();
```

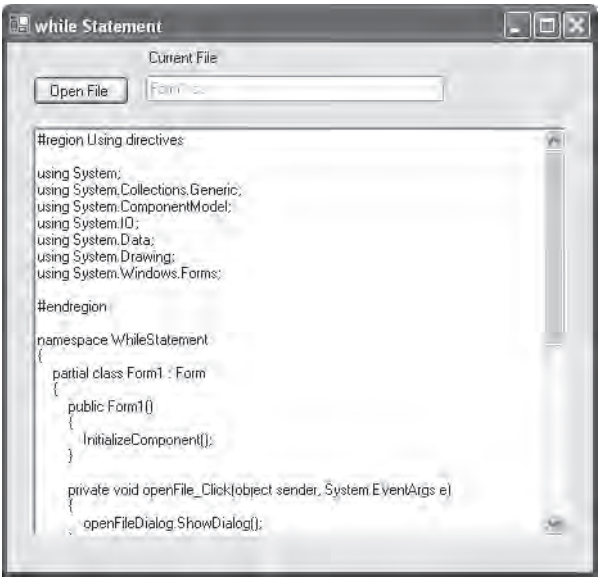
בדרך זו, הביטוי הבוליאני שבתחילת הלולאה מבצע את האתחול וגם את העדכון. יש קריאה לשיטה **ReadLine**, והערך המוחזר ממנה מוצב במשתנה **line**. אולם, משפט ההצבה מניב למעשה ערך - ערך המשתנה שקולט את הנתון. לפיכך, אפשר להשוות את תוצאת ביטוי ההצבה (assignment expression) באמצעות אופרטור יחס (relation operator) ולקבל תוצאה בוליאנית. בדוגמה זו, אם הערך המוצב הוא **null**, ערך ביטוי ההצבה יהיה **null**, ותוצאת ההשוואה ל-**null** תהיה אמת.

11. בתפריט Debug בחר Start Without Debugging.

12. לחץ Open File.

13. פתח את התיקייה \My Documents\Microsoft Press\Visual CSharp Step by Step\Chapter 5\WhileStatement\WhileStatement. בחר את הקובץ Form1.cs, ולחץ Open.

הפעם תכולת הקובץ הנבחר תוצג בתיבת הטקסט.



14. בתיבת הטקסט, אתר את השיטה **openFileDialog_FileOk**. ודא שהשיטה מכילה את הקוד שזה עתה הוספת.

15. סגור את התוכנית כדי לחזור לסביבת הפיתוח של Visual Studio 2005.

כתיבת משפטי for

המשפטים מסוג **while** נכתבים בדרך כלל על פי התבנית הזו:

```
initialization
while (Boolean expression)
{
    statement
    update control variable
}
```

משפט **for** (**עבור**) מאפשר לכתוב גרסה פורמאלית יותר לסוג זה של מבנה, על ידי שילוב של האתחול, הביטוי הבוליאני והעדכון (המרכיבים "המתחזקים" את הלולאה). לולאת for שימושית ביותר, כי היא אינה "מאפשרת" לשכוח את שלושת המרכיבים הללו. להלן התחביר של משפט מסוג **for**:

```
for (initialization; Boolean expression; update control variable)
    statement
```

את לולאת **while** שהוצגה קודם לכן, כדי להציג את הערכים השלמים שבין 0 ל-9, ניתן להציג כלולאת **for**:

```
for (int i = 0; i != 10; i++)
{
    Console.WriteLine(i);
}
```

האתחול מבוצע בתחילת הלולאה. כאשר הביטוי הבוליאני הוא אמת, המשפט יופעל. לאחר מכן משתנה הבקרה מעודכן והביטוי הבוליאני נבדק שוב. אם התנאי עודנו אמת, המשפט יופעל שוב, משתנה הבקרה יעודכן, הביטוי ייבדק, וחוזר חלילה.

שים לב שהאתחול מתבצע פעם אחת בלבד; שהמשפט בגוף הלולאה מופעל תמיד **לפני** עדכון המשתנה; ושהעדכון מתבצע **לפני** בדיקת הביטוי הבוליאני.

ניתן להשמיט כל אחד משלושת הרכיבים שבמשפט **for**. אם תשמיט את הביטוי הבוליאני ברירת המחדל תהיה אמת. משפט **for** שלהלן יפעל שוב ושוב לעד (לולאה אינסופית), כי אין בו מרכיב סיום:

```
for (int i = 0; ; i++)
{
    Console.WriteLine("Help, somebody stop me!");
}
```

השמטה של האתחול והעדכון יצרו לולאת **while** בתחביר משונה:

```
int i = 0;
for (; i != 10; )
{
    Console.WriteLine(i);
    i++;
}
```

הערה

במשפטי **for**, האתחול, הביטוי הבוליאני ועדכון משתנה הבקרה צריכים להיות תמיד מופרדים בנקודה-פסיק (;).

במידת הצורך, ניתן להוסיף מספר אתחולים למשתנים שונים ומספר עדכונים עבור הלולאה (אבל חייב להיות ביטוי בוליאני אחד בלבד). כדי לעשות זאת, הפרד את האתחולים והעדכונים השונים בפסיקים, כמודגם להלן:

```
for (int i = 0, j = 10; i <= j; i++, j--)
{
    ...
}
```

טיפ

רצוי לסמן בלוק משפטים ברור בגוף משפטים מסוג **for**, **while** או **if**, אפילו כאשר הבלוק מכיל משפט אחד בלבד. הדבר יקל על הוספת משפטים לבלוק בשלב מאוחר יותר. אם לא תעשה זאת, בעת הוספת משפטים תצטרך לזכור להוסיף את סימון הסוגריים המסולסלים של הבלוק.

תחום ההכרזה של משפט for

ייתכן שהבחנת כבר בכך שניתן להכריז על משתנה בשלב האתחול של משפט **for**. תחום ההכרזה של משתנה זה מוגבל לגוף משפט **for**, והוא ייעלם כאשר משפט **for** יסתיים. לכלל זה שתי השלכות חשובות. ראשית, אינך יכול להשתמש במשתנה לאחר סיום משפט **for**, כי אינך נמצא בתחום ההכרזה שלו. הנה דוגמה:

```
for (int i = 0; i != 10; i++)
{
    ...
}
Console.WriteLine(i); // compile time error
```

שנית, ניתן לכתוב שני משפטי **for** רצופים המשתמשים באותו שם משתנה, כי לכל אחד מהמשתנים יש תחום הכרזה נפרד. הנה דוגמה:

```
for (int i = 0; i != 10; i++)
{
    ...
}
for (int i = 0; i != 20; i += 2) // okay
{
    ...
}
```

כתיבת משפטי do

משפטי **while** ו-**for** בודקים את הביטוי הבוליאני בתחילת הלולאה. משמעות הדבר היא שאם הביטוי שקרי (false) בבדיקה הראשונה, גוף הלולאה לא יופעל כלל. משפטי do שונים מהבחינה הזו. הביטוי הבוליאני נבדק **אחרי** כל חזרה, ולכן גוף הלולאה מופעל **לפחות פעם אחת**.

זהו התחביר של משפטי **do** (במשמעות "ביצוע"). חשוב לא לשכוח את הנקודה-פסיק (;) הסוגרת את המשפט:

```
do
    statement
while (booleanExpression);
```

קבץ את המשפטים בבלוק כאשר גוף הלולאה מורכב מיותר ממשפט אחד. להלן גרסה של הדוגמה האחרונה אשר מציגה במסך את המספרים השלמים בין 0 ל-9, כפי שמבוצעת על ידי משפט **do**:

```
int i = 0;
do
{
    Console.WriteLine(i);
    i++;
}
while (i != 10);
```

שים לב שפעולת ההדפסה, ולמעשה ההצגה, תבוצע בפעם הראשונה ללא קשר לתנאי, מפני שבדיקת התנאי נעשית לאחר הביצוע הראשון.

משפטי break ו-continue

בפרק 4 למדת כיצד משפט **break** (**שבירה**) משמש ליציאה ממשפט **switch**. משפט **break** יכול לשמש גם ליציאה ממשפט איטרציה. שבירת הרצף של לולאה גוררת יציאה מיידית ממנה והפעלת המשפט הראשון שלאחריה. אין הפעלה חוזרת של העדכון, וגם לא של תנאי החזרה של הלולאה.

משפט **continue** (המשך), לעומת זאת, גורם לתוכנית לחזור מייד לביצוע נוסף של הלולאה (לאחר בדיקה חוזרת של הביטוי הבוליאני). להלן גרסה של הדוגמה האחרונה אשר מציגה במסך את המספרים השלמים בין 0 ל-9, והפעם ־ על ידי שימוש במשפטי **break** ו-**continue**.

```
int i = 0;
while (true)
{
    Console.WriteLine("continue " + i);
    i++;
    if (i != 10)
        continue;
    else
        break;
}
```

הקוד מסורבל ביותר. ההנחיות למתכנתים ממליצות להשתמש במשפטי **continue** בזהירות רבה, או לא להשתמש בהם כלל, כי לעיתים קרובות הם גוררים קוד מסורבל וקשה למעקב.

בתרגיל הבא עליך לכתוב משפט **do** להמרת מספר למחרוזת המייצגת אותו.

דוגמה למשפט do

1. בסביבת הפיתוח Visual Studio 2005, פתח את הפרויקט DoStatement שבתיקייה My Documents\Microsoft Press\Visual CSharp Step by Step\Chapter 5\DoStatement.
2. בתפריט Debug בחר Start Without Debugging.

   סביבת הפיתוח תבנה ותפעיל את היישום Windows.

   היישום יציג טופס הכולל שתי תיבות טקסט ולחצן Show Steps.

   אם תקליד מספר חיובי (האלגוריתם המשמש את היישום אינו פועל על מספרים שליליים) בתיבת הטקסט העליונה ותלחץ Show Steps, יוצגו השלבים ביצירת מחרוזת המייצגת את המספר.

הערה

זו דוגמה להמרת מספר למחרוזת באמצעות לולאת **do**. סביבת הפיתוח .NET Framwork כוללת את השיטה **Convert.ToString** המבצעת את אותה פעולה, ולמעשה מומלץ להשתמש בה כדי לבצע פעולה זו.

3. בתור דוגמה, הקלד 2693 בתיבת הטקסט העליונה, ולחץ Show Steps.

תיבת הטקסט התחתונה תציג את הצעדים ליצירת מחרוזת המייצגת את המספר 2693:

4. סגור את החלון וחזור לסביבת הפיתוח Visual Studio 2005.

5. הצג את הקוד של Form1.cs בחלון Code and Text Editor.

6. אתר את השיטה **showSteps_Click**. אשר מופעלת כאשר המשתמש לוחץ Show Steps.

היא מכילה את המשפטים הבאים:

```
int amount = System.Int32.Parse(number.Text);

steps.Text = “”;
string current = “”;
do
{
    int digitCode = ‘0’ + amount % 10;
    char digit = Convert.ToChar(digitCode);
    current = digit + current;
    steps.Text += current + “\r\n”;
    amount /= 10;
}
while (amount != 0);
```

הערה

המשמעות של \r היא החזרת גרר (carriage return, זו למעשה פעולת Enter, והכינוי הוא מתקופת מכונות הכתיבה...). בעת הזנת **טקסט בתיבת טקסט** מרובת שורות, עליך להשתמש בסימון החזרת גרר ובשורה חדשה (newline) כדי לעבור לשורה חדשה ולהחזיר את הסמן לתחילת השורה; אחרת, כל הטקסט יופיע כשורה אחת.

המשפט הראשון ממיר את המחרוזת שבמאפיין **Text** של תיבת הטקסט **number** לערך שלם (**integer**), באמצעות השיטה **Parse** מהמחלקה **System.Int32**:

```
int amount = System.Int32.Parse(number.Text);
```

המשפט השני מוחק את הטקסט המוצג בתיבת הטקסט התחתונה (שנקראת **steps**) על ידי הצבת מחרוזת ריקה במאפיין **text** של התיבה:

```
steps.Text = “”;
```

המשפט השלישי מכריז על משתנה **current** מסוג **string** ומציב בו מחרוזת ריקה:

```
string current = “”;
```

העבודה האמיתית בשיטה זו מתבצעת על ידי המשפט **do** שמתחיל במשפט הרביעי:

```
do
{
    ...
}
while (amount != 0);
```

האלגוריתם משתמש באריתמטיקה של מספרים שלמים ובאופרטור השארית (modulus) כדי לחלק את המשתנה **amount** ב-10. השארית לאחר כל פעולת חילוק היא למעשה התו הבא במחרוזת שנבנית. בסופו של דבר, הערך של **amount** מגיע ל-0, והלולאה נפסקת. שים לב שגוף הלולאה חייב לפעול פעם אחת לפחות. התנהגות זו מתאימה בדיוק לנסיבות, כי גם המספר 0 הוא בעל ספרה אחת.

המשפט הראשון בלולאת **do** הוא:

```
int digitCode = ‘0’ + amount % 10;
```

המשפט מכריז על משתנה **digitCode** מסוג **int** ומאתחל אותו לתוצאת הביטוי הבא:

```
‘0’ + amount % 10
```

ביטוי זה דורש הסבר! הערך ‘0’ הוא התו אפס. במערך התווים המשמש את Windows, תו זה מקביל לערך השלם 48 (לכל תו יש קוד משלו; אשר מייצג מספר שלם). הקוד עבור התו "1" הוא 49, עבור התו "2" 50 וכן הלאה.

הערך של הביטוי amount % 10 שווה לשארית הנותרת מהחלוקה של **amount** ב-10. לדוגמה, אם הערך של **amount** הוא 2693, אזי 2693 % 10 שווה ל-3 (כי 2693 חלקי 10 שווה ל-269 ושארית 3). לכן, אם **amount** שווה ל-2693, הביטוי amount % 10 + '0' זהה לביטוי 3 + '0', כלומר 51. זהו הקוד עבור התו '3' (האופרטור + ממיר את התו '0' לערך 48 כדי שיהיה ניתן לבצע את החישוב).

המשפט השני בלולאת **do** הוא:

```
char digit = Convert.ToChar(digitCode);
```

משפט זה מכריז על משתנה **digit** מסוג **char** ומאתחל אותו לערך המוחזר מהקריאה לשיטה **Convert.ToChar(digitCode)**. קריאה זו מחזירה את הערך **char** המקביל לקוד התו שהוצב בתור ארגומנט. במילים אחרות, הערך של Convert.ToChar('0' + 3) שווה לערך של '3'.

המשפט השלישי בלולאת **do** הוא:

```
current = digit + current;
```

המשפט מוסיף את המשתנה **digit** מסוג **char** אל תחילת המחרוזת **current** מסוג string. לא ניתן להחליף את המשפט הזה במשפט current += digit, כי בדרך זו התו יתווסף לסוף המחרוזת.

המשפט הרביעי בלולאת **do** הוא:

```
steps.Text += current + "\r\n";
```

המשפט מוסיף את המחרוזת שנבנתה אל תיבת הטקסט.

המשפט האחרון בלולאת **do** הוא:

```
amount /= 10;
```

למשפט זה משמעות זהה לזו של המשפט amount = amount / 10;. אם הערך של **amount** הוא 2693, ערכו לאחר הפעלת המשפט יהיה 269. שים לב שכל חזרה על הלולאה **do** מביאה להשמטת הספרה האחרונה מהמשתנה **amount** והוספה שלה לתחילת המחרוזת current.

בתרגיל האחרון לפרק זה, תשתמש בתוכנית הניפוי (debugger) של Visual Studio 2005 כדי לסקור את משפט **do** שלעיל בזמן פעולתו, וכך להבין כיצד הוא פועל.

סקירת ביצוע של משפט do

1. בחלון Code and Text Editor, אתר את השיטה showSteps_Click.
2. העבר את הסמן למשפט הראשון בשיטה showSteps_Click.
   להלן המשפט הראשון:

```
int amount = System.Int32.Parse(number.Text);
```

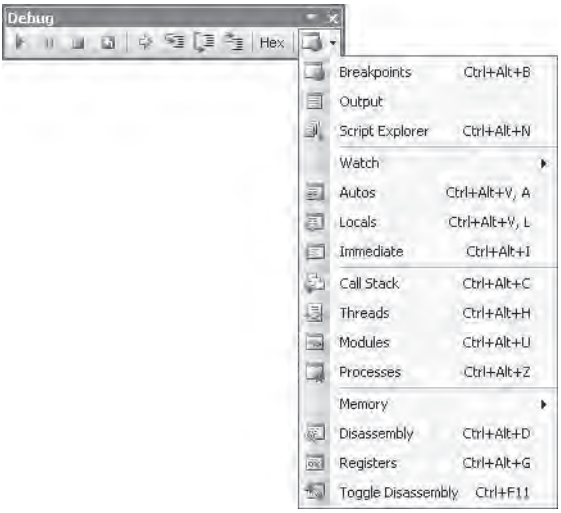
3. לחץ לחיצה ימנית על המשפט הראשון ובתפריט הקיצור בחר Run To Cursor.

סביבת הפיתוח תבנה ותפעיל את היישום.

4. בטופס שיופיע, הקלד 2693 בתיבת הטקסט העליונה ולחץ Show Steps.

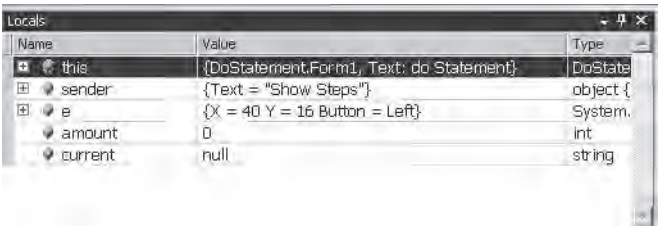
התוכנית תעצור ותחזור למצב הניפוי של סביבת הפיתוח. החץ הצהוב בצד השמאלי של החלון Code and Text Editor מציין את המשפט הנוכחי.

5. הצג את סרגל הכלים Debug אם אינו מוצג (בתפריט View, Toolbars סמן את Debug). בסרגל הכלים Debug לחץ על חץ התפריט הנפתח, ואז יופיע התפריט הבא:



6. בחר Locals מהתפריט.

חלון Locals ייפתח ויציג את השם, הערך והסוג של המשתנים המקומיים בשיטה הנוכחית, כולל המשתנה המקומי **amount**. שים לב שהערך של **amount** עומד כרגע על אפס:



7. בסרגל הכלים Debug לחץ על Step Into.

המנפה יפעיל את המשפט הבא:

```
int amount = System.Int32.Parse(number.Text);
```

ערך המשתנה **amount** בחלון Locals ישתנה ל-2693 והחץ הצהוב יעבור למשפט הבא.

8. לחץ Step Into.

המנפה יפעיל את המשפט:

```
steps.Text = “”;
```

משפט זה אינו משפיע על החלון Locals כי steps הוא שדה בטופס, ולא משתנה מקומי. החץ הצהוב יעבור למשפט הבא.

9. לחץ Step Into.

המנפה יפעיל את המשפט:

```
string current = “”;
```

החץ הצהוב יעבור לסוגר המסולסל השמאלי שבתחילת הלולאה **do**.

10. לחץ Step Into.

החץ הצהוב יעבור למשפט הראשון בתוך לולאת **do**. אשר בה שני משתנים מקומיים משלה, **digitCode** ו-**digit**. שים לב שהמשתנים החדשים הופיעו בחלון Locals ושערכו של המשתנה **digitCode** הוא אפס.

11. לחץ Step Into.

המנפה יפעיל את המשפט:

```
int digitCode = ‘0’ + amount % 10;
```

ערך המשתנה **digitCode** בחלון Locals ישתנה ל-51. הסיבה לכך היא שערך הביטוי amount % 10 הוא 3 (ערך **amount** הוא 2693), והקוד עבור התו ‘3’ הוא 51 .

12. לחץ Step Into.

המנפה יפעיל את המשפט:

```
char digit = Convert.ToChar(digitCode);
```

הערך של **digit** ישתנה ל-‘3’ בחלון Locals, אשר מציג ערכים מסוג **char** גם באמצעות הערך הנומרי המייצג אותם (במקרה זה 51) וגם באמצעות התו עצמו (‘3’). החץ הצהוב יעבור למשפט הבא שבלולאה **do**.

שים לב שבחלון Locals, ערך המשתנה **current** הוא “” (כלומר, ריק).

13. לחץ Step Into.

המנפה יפעיל את המשפט:

```
current = current + digit;
```

הערך של **current** ישתנה ל-"3" בחלון Locals.

14. לחץ Step Into.

המנפה יפעיל את המשפט:

```
steps.Text += current + "\r\n";
```

המשפט יציג את הטקסט "3" בתיבת הטקסט steps, כשלאחריו תו החזרת גרר ותו שורה חדשה, כדי שהפלט הנוסף יוצג בשורה חדשה בתיבת הטקסט.

בחלון Locals ערך **amount** עודנו 2693.

15. לחץ Step Into.

המנפה יפעיל את המשפט:

```
amount /= 10;
```

הערך של **amount** ישתנה ל-269 בחלון Locals. החץ הצהוב יעבור אל הסוגר המסולסל הימני שבסוף לולאת **do**.

16. לחץ Step Into.

החץ הצהוב יעבור למשפט **while**.

17. לחץ Step Into.

המנפה יפעיל את המשפט:

```
while (amount != 0);
```

הערך של amount הוא 269, והביטוי **269 != 0** הוא **true**, ולכן הלולאה **do** צריכה לבצע חזרה נוספת. החץ הצהוב יחזור לסוגר המסולסל השמאלי שבתחילת הלולאה **do**.

18. לחץ Step Into.

החץ הצהוב יחזור שוב למשפט הראשון שבלולאה **do**.

19. לחץ Step Into 22 פעמים נוספות וצפה בערכי המשתנים המקומיים משתנים בחלון Locals.

הערך של **amount** עומד כרגע על אפס, והערך של **current** הוא "2693". החץ הצהוב מצביע על תנאי החזרה של הלולאה **do**.

```
while (amount != 0);
```

הערך של **amount** הוא 0, ולכן הביטוי **amount != 0** הוא **false**, וכתוצאה - הלולאה **do** צריכה להסתיים.

20. לחץ Step Into.

המנפה יפעיל את המשפט:

```
while (amount != 0);
```

כמצופה, לולאת **do** נפסקת והחץ הצהוב מצביע על הסוגר הימני שבסוף השיטה **showSteps_Click**.

21. לחץ Continue.

הטופס יופיע ויציג את ארבעת השלבים המשמשים ליצירת המחרוזת המייצגת את המספר 2693: "3", "93", "693" ו-"2693".

22. סגור את הטופס כדי לחזור לסביבת הפיתוח Visual Studio 2005.

מזל טוב! לראשונה כתבת משפטי **while** ו-**do** בעלי משמעות והפעלת את תוכנית הניפוי של Visual Studio 2005 כדי לסקור את פעולת משפט **do**.

☯ **אם ברצונך להמשיך לפרק הבא**

השאר את Visual Studio 2005 פעילה ועבור לפרק 6.

☯ **אם ברצונך לסגור את** Visual Studio 2005

בתפריט File בחר Exit. אם מופיעה תיבת דו-שיח Save, לחץ Yes.

המשימה	צריך
להוסיף סכום למשתנה.	להשתמש באופרטור הוספה מורכבת. לדוגמה: variable += amount;
לחסר סכום ממשתנה.	להשתמש באופרטור החסרה מורכבת. לדוגמה: variable -= amount;

להפעיל משפט אחד או יותר כל עוד תנאי מסוים מתקיים.	להשתמש במשפט **while**. לדוגמה:
	`int i = 0;`
	`while (i != 10)`
	`{`
	`  Console.WriteLine(i);`
	`  i++;`
	`}`
	או לחילופין להשתמש במשפט **for**:
	`for (int i = 0; i != 10; i++)`
	`{`
	`  Console.WriteLine(i);`
	`}`

להפעיל משפטים ברצף פעם אחת או יותר (לפחות פעם אחת).	להשתמש במשפט **do**. לדוגמה:
	`int i = 0;`
	`do`
	`{`
	`  Console.WriteLine(i);`
	`  i++;`
	`}`
	`while (i != 10);`

פרק 6

ניהול שגיאות וחריגים

בסיום פרק זה, תוכל:

- לטפל בחריגים באמצעות משפטי **try**, **catch** ו-**finally**.
- לשלוט בגלישה של מספרים שלמים באמצעות מילות המפתח checked ו-unchecked.
- ליצור חריגים יזומים משיטה באמצעות מילת המפתח throw.
- להבטיח שהקוד לא יפסיק לפעול, למרות היווצרות חריג, באמצעות בלוק **finally**.

כעת אתה כבר מכיר את הגרעין החשוב של משפטי #C המאפשרים לך לקרוא ולכתוב שיטות, להכריז על משתנים, להשתמש באופרטורים כדי ליצור ערכים, לכתוב משפטי **if** ו-**switch** כדי להפעיל קוד באופן סלקטיבי ולכתוב משפטי **while**, **for** ו-**do** כדי להפעיל קוד בצורה מחזורית. הפרקים הקודמים לא התייחסו לאפשרות (או הסבירות) שדברים עשויים להשתבש. קשה מאוד להבטיח שמקטע של קוד יפעל כמצופה. תקלות עלולות להיות עקב מספר רב של סיבות, חלקן מעבר לשליטתך בתור מתכנת. כל יישום שאתה כותב חייב לכלול את היכולת לזהות ולטפל בשגיאות באופן אלגנטי. בפרק זה, הפרק האחרון בחלק א' של הספר, תלמד כיצד #C זורק חריגים כדי לסמן מקרה של שגיאה, וכיצד להשתמש במשפטים **try**, **catch** ו-**finally** כדי לאתר את השגיאות אשר אותם חריגים מייצגים ולטפל בהן. בסופו של פרק זה יהיה ברשותך בסיס מוצק לעבודה עם #C. בסיס זה יאפשר לך להמשיך ולהרחיב את יכולותיך בחלק ב'.

טיפול בשגיאות

אחת מעובדות החיים היא שדברים רעים קורים לעיתים. לצמיגים יש תקרים, סוללות מתרוקנות, מברגים לעולם אינם נמצאים במקום שהנחת אותם ומשתמשי מחשב לעולם אינם מתנהגים כפי שצפית. שגיאות עשויות להתרחש כמעט בכל שלבי פעולה של התוכנית. איך ניתן לזהות ולהתגבר על שגיאות אלו?

כדי להשיב על שאלה זו, עליך להכיר את **החריגים** (**exceptions**).

בדיקת קוד ותפיסת חריגים

שפת #C מקלה על ההפרדה בין קוד המבצע את פקודות התוכנית עצמה לבין קוד המטפל בשגיאות, באמצעות שימוש בחריגים ובמטפלי חריגים (exception handlers). כדי לכתוב תוכניות אשר מנצלות את היכולות של החריגים, עליך לעשות שני דברים:

1. כתוב את קוד התוכנית שלך בתוך בלוק **try** (**ניסוי**, זו מילת מפתח). כאשר הקוד פועל, הוא ינסה להפעיל את כל המשפטים שבתוך הבלוק **try**, ואם אף אחד מהם אינו מניב חריג, הוא יפעיל את כולם, בזה אחר זה, עד לסוף התוכנית. אולם, במקרה של שגיאה, ההפעלה מדלגת מתוך הבלוק **try** אל מטפל השגיאות **catch**.
2. כתוב מטפל אחד או יותר מסוג **catch** (זו מילת מפתח) מיד לאחר הבלוק **try** שתפקידו לאתר שגיאות ולזרוק (**throw**) חריגים. אם אחד מהמשפטים בתוך הבלוק **try** יוצר שגיאה, סביבת ההרצה (runtime) תיצור חריג ותזרוק אותו. לאחר מכן סביבת ההרצה תבדוק את המטפלים מסוג **catch** שמופיעים לאחר הבלוק **try** ותעביר את השליטה ישירות למטפל המתאים. מטפלי **catch** מתוכננים בצורה שתאפשר להם ללכוד חריגים ספציפיים, עובדה המאפשרת ליצור מספר מטפלים שונים עבור סוגים שונים של שגיאות העשויות לקרות.

הדוגמה הבאה משתמשת בבלוק **try** כדי לנסות ולהמיר מספר שדות טקסט לערכים שלמים (integer), לקרוא לשיטה לחישוב ערך ולהדפיס את התוצאה בשדה text. כדי להמיר מחרוזת למספר שלם המחרוזת צריכה להכיל ייצוג תקין ולא רצף שרירותי של תווים. אם המחרוזת מכילה תווים שאינם מתאימים, השיטה **Int32.Parse** תזרוק **FormatException**, והמטפל **catch** המתאים יופעל. בסיום פעולת המטפל **catch**, התוכנית תמשיך לפעול מהמשפט הראשון לאחר המטפל:

```
try
{
   int leftHandSide = Int32.Parse(leftHandSideOperand.Text);
   int rightHandSide = Int32.Parse(rightHandSideOperand.Text);
   int answer = doCalculation(leftHandSide, rightHandSide);
   result.Text = answer.ToString();
}
catch (FormatException fEx)
{
   // Handle the exception
   ...
}
```

טיפול בחריגים

המטפל **catch** משתמש בתחביר דומה לזה המשמש פרמטרים של שיטות לפירוט החריג שיש לאתר. בדוגמה האחרונה, כאשר **FormatException** נזרק, המשתנה **fEx** מתמלא באובייקט המכיל את פרטי החריג. לסוג **FormatException** יש מספר שדות שניתן לבדוק כדי לקבוע בדיוק את הגורם לחריג. חלק גדול מהשדות הללו משותפים לכל החריגים. לדוגמה, השדה **Message** מכיל טקסט המתאר את השגיאה שגרמה לחריג. באפשרותך להשתמש במידע זה בעת הטיפול בחריג, תיעוד הפרטים ביומן (log file) או בהעברת הודעת מידע המבקשת מהמשתמש לנסות שוב.

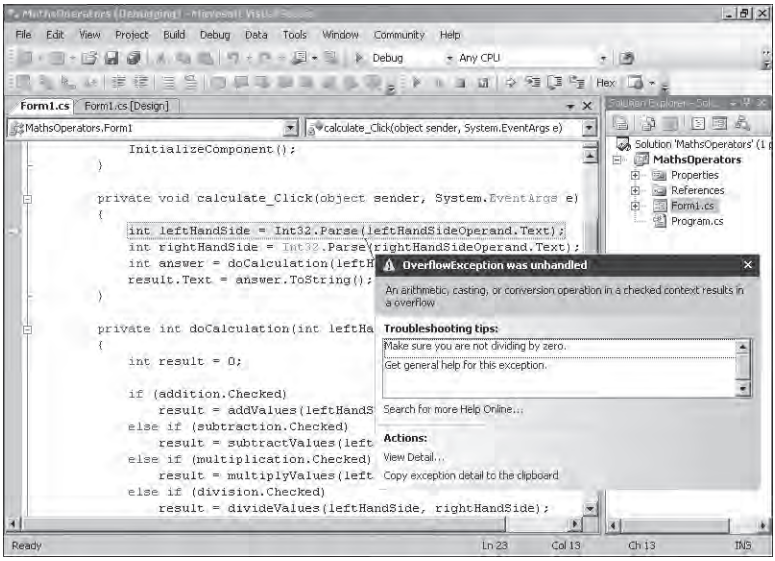
חריגים לא מטופלים

מה קורה כאשר בלוק **try** זורק חריג (throws an exception), אולם אין מטפל **catch** מתאים כדי לקלוט ולפענח אותו? בדוגמה הקודמת, ייתכן שהשדה **leftHandSideOperand** יכיל מחרוזת המייצגת מספר שלם תקין, אולם המספר השלם אינו בתחום המספרים השלמים הנתמכים על ידי C# (לדוגמה, "2147483648"). במקרה זה, המשפט **Int32.Parse** יזרוק **OverFlowException**, אשר לא ייתפס על ידי המטפל **catch** כיוון שהוא מטפל ב-**FormatException**. אם הבלוק **try** מהווה חלק משיטה, היא תיפסק ותחזור לשיטה שקראה לה. אם השיטה הקוראת משתמשת גם היא בבלוק **try**, סביבת ההרצה CLR (Common Language Runtime) תנסה לאתר מטפל **catch** מתאים לאחר הבלוק **try** ולהפעיל אותו. אם השיטה שביצעה את הקריאה אינה משתמשת בבלוק **try**, או אם עדיין לא נמצא מטפל **catch** מתאים, השיטה שביצעה את הקריאה תסתיים ותחזור לגורם שקרא לה, והתהליך יחזור על עצמו. אם בסופו של דבר יימצא מטפל **catch** מתאים, המטפל יופעל והתוכנית תמשיך לפעול מהמשפט הראשון לאחר המטפל **catch** בשיטה אשר תפסה את החריג.

חשוב

שים לב שלאחר תפיסת חריג, פעולת התוכנית ממשיכה מהשיטה שמכילה את הבלוק **catch** אשר תפס את החריג. השליטה **אינה** חוזרת אל השיטה אשר זרקה את החריג.

אם סביבת ההרצה (CLR) לא מצאה מטפל catch מתאים, התוכנית תיפסק עקב חריג לא מטופל. אם התוכנית פועלת במצב Debug של סביבת הפיתוח Visual Studio 2005 (כלומר שבחרת Start Debugging בתפריט Debug כדי להפעיל את היישום), תיבת הדו-שיח האינפורמטיבית הבאה תופיע על המסך:



שימוש במספר מטפלי catch

הדיון שלעיל מדגיש את העובדה ששגיאות שונות זורקות חריגים שונים, המייצגים סוגים שונים של תקלות. כדי להתמודד עם מצבים אלה, עליך לכתוב מספר מטפלי **catch**, בזה אחר זה, כמו בדוגמה זו:

```
try
{
   int leftHandSide = Int32.Parse(leftHandSideOperand.Text);
   int rightHandSide = Int32.Parse(rightHandSideOperand.Text);
   int answer = doCalculation(leftHandSide, rightHandSide);
   result.Text = answer.ToString();
}
catch (FormatException fEx)
{
   //...
}
catch (OverflowException oEx)
{
   //...
}
```

תפיסת חריגים מרובים

מנגנון תפיסת החריגים של סביבת ההרצה (CLR) מקיף למדי. סוגים רבים של חריגים מוגדרים תחת NET Framework., וכל תוכנית שתכתוב תוכל לזרוק את רובם. הסבירות שתכתוב מטפל **catch** לכל חריג אשר הקוד שלך עשוי לזרוק, היא נמוכה מאוד. אם כך, כיצד ניתן להבטיח שכל החריגים ייתפסו ויטופלו?

התשובה לשאלה זו טמונה בקשר שבין החריגים השונים. החריגים מחולקים למחלקות/משפחות אשר נקראות היראכיות תורשתיות (inheritance hierarchies) (על תורשתיות נלמד בפרק 12). גם **FormatException** וגם **OverFlowException** שייכים, יחד עם כמה חריגים נוספים, למשפחה בשם **SystemException**. במקום לתפוס כל אחד מהחריגים הללו באופן נפרד, ניתן ליצור מטפל אשר תופס את סוג החריגים במשפחה **SystemException**. המשפחה **SystemException** עצמה, שייכת למשפחת-על בשם **Exception**. אם אתה לוכד את **Exception**, המטפל לוכד כל חריג אפשרי אשר עשוי לקרות.

הערה

המשפחה **Exception** כוללת מגוון חריגים, אשר רבים מהם משמשים בחלקים שונים של סביבת ההרצה (CLR). חלקם אזוטריים למדי, אולם כדאי בכל זאת לדעת כיצד ללכוד אותם.

הדוגמה הבאה ממחישה כיצד לתפוס את כל החריגים חברי המשפחה SystemException:

```
try
{
   int leftHandSide = Int32.Parse(leftHandSideOperand.Text);
   int rightHandSide = Int32.Parse(rightHandSideOperand.Text);
   int answer = doCalculation(leftHandSide, rightHandSide);
   result.Text = answer.ToString();
}
catch (Exception ex) // this is a general catch handler
{
   //...
}
```

טיפ

אם ברצונך לתפוס את **Exception**, תוכל להשמיט את שמו (Exception ex) מהמטפל **catch**, כי זהו חריג ברירת מחדל.

```
catch { // ... }
```

אולם אין זה מומלץ. פריט החריג שמועבר למטפל **catch** מכיל מידע שימושי על אודות החריג, ובגרסה זו של המטפל **catch** אין גישה למידע הזה.

כעת נותרה שאלה אחת אחרונה: מה קורה אם חריג כלשהו מתאים ליותר ממטפל **catch** אחד מבין המטפלים הנמצאים בסוף הבלוק **try**? אם תתפוס את **FormatException** ואת **Exception** בשני מטפלים שונים, איזה מבין השניים יופעל?

כאשר מופיע חריג (when an exception occurs), מופעל המטפל הראשון המתאים לחריג, וסביבת ההרצה מתעלמת מהמטפלים האחרים. משמעות הדבר היא שאם תציב מטפל עבור **Exception** לפני המטפל עבור **FormatException**, המטפל עבור **FormatException** לא יופעל לעולם. לפיכך, יש להציב את המטפלים הפרטניים יותר מעל המטפלים הכללים, לאחר הבלוק **try** כמובן. אם אף לא אחד מהטפלים הפרטניים יתפוס את החריג, יתפוס אותו בוודאי המטפל **catch** הכללי.

בתרגיל הבא עליך לכתוב בלוק **try** ולתפוס חריג.

כתוב משפט try/catch

1. הפעל את Visual Studio 2005.

2. פתח את הפתרון **MathsOperators** שבתיקייה My Documents\Microsoft Press\Visual CSharp Step By Step\Chapter 6\MathsOperators.

זו גרסה של התוכנית שעסקת בה לראשונה בפרק 2 ואשר שימשה להדגמת האופרטורים האריתמטיים השונים.

3. בתפריט Debug בחר Start Without Debugging.

הערה

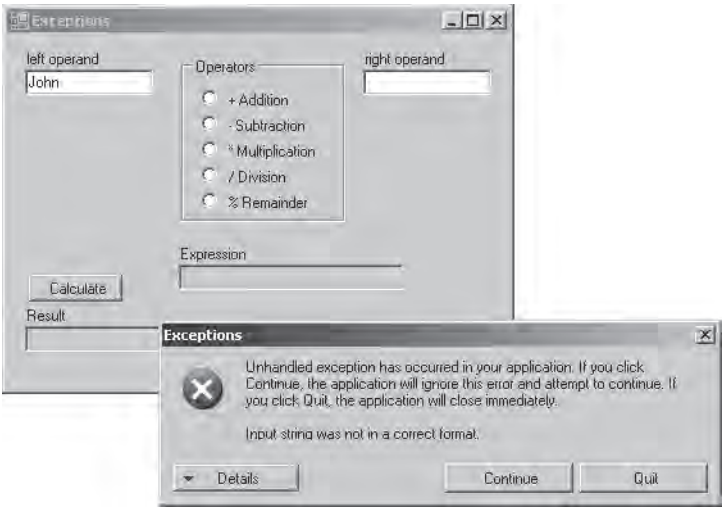
כאשר תפעיל את היישום במצב Debug, הוא ייכנס למנפה במקרה של חריג לא מטופל. אין לנו עניין שהדבר יקרה בדוגמה זו, ולכן צרין לבחור באפשרות Start Without Debugging.

סביבת הפיתוח Visual Studio 2005 תבנה ותפעיל את היישום Windows. הטופס Exceptions יוצג במסך.

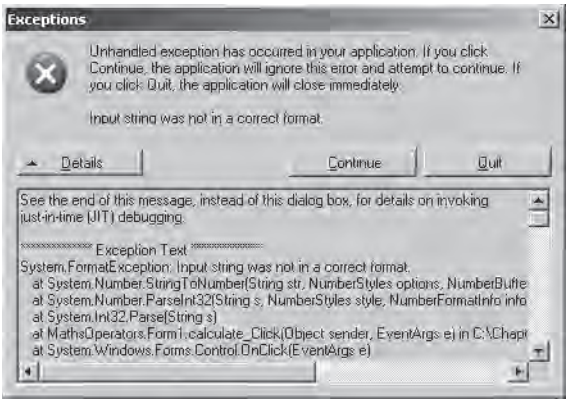
כעת הכנס במתכוון טקסט שאינו חוקי בתיבת הטקסט של האופרנד השמאלי. פעולה זו תדגים את חוסר היציבות בגרסה זו של התוכנה.

4. בתיבת הטקסט של האופרנד השמאלי הקלד **John** ולחץ Calculate.

תיבת הדו-שיח תודיע על חריג לא-מטופל. הטקסט שהזנת בתיבת הטקסט של האופרנד השמאלי הביאה לקריסת התוכנה.



5. בתיבת הדו-שיח Exceptions לחץ Details כדי להציג את המידע על החריג.



על פי השורות הראשונות של הטקסט ניתן לראות שהחריג נזרק על ידי הקריאה ל-**Int32.Parse** שבתוך השיטה **calculate_Click**.

6. לחץ Quit כדי לסגור את תיבת הדו-שיח Exceptions ולחזור לסביבת הפיתוח.

7. הצג את הקוד של הקובץ Form1.cs בחלונית Code.

8. אתר את השיטה **calculate_Click**. הוסף בלוק **try** סביב ארבעת המשפטים שבתוך השיטה. הקוד נראה כעת כך:

```
try
{
    int leftHandSide = Int32.Parse(leftHandSideOperand.Text);
    int rightHandSide = Int32.Parse(rightHandSideOperand.Text);
    int answer = doCalculation(leftHandSide, rightHandSide);
    result.Text = answer.ToString();
}
```

9. הוסף בלוק **catch** לאחר הקוד **try**, כפי שמוצג כאן:

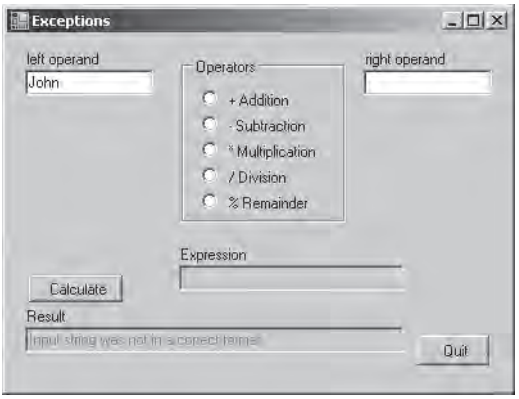
```
catch (FormatException fEx)
{
   result.Text = fEx.Message;
}
```

המטפל **catch** תופס את החריג **FormatException** שנזרק על ידי **Int32.Parse**, ולאחר מכן מציב את הטקסט **Message** במאפיין **Text** של תיבת הטקסט **Result** אשר בתחתית הטופס.

10. בתפריט Debug בחר Start Without Debugging.

11. בתיבת הטקסט של האופרנד השמאלי, הקלד **John** ולחץ Calculate.

המטפל **catch** מצליח לתפוס את החריג **FormatException** וההודעה "Input string was not in a correct format" תופיע בתיבת הטקסט Result. כעת היישום יציב יותר.



12. לחץ Quit כדי לחזור לסביבת הפיתוח של Visual Studio 2005.

שימוש באריתמטיקה של מספרים שלמים

Unchecked–1 Checked

בפרק 2 למדת להשתמש באופרטורים של אריתמטיקה בינארית כגון + ו-* עם סוגי נתונים פרימיטיביים (primitive data types) כגון **int** ו-**double**. גם למדת שסוגי מידע פרימיטיבי הם בעלי גודל קבוע. לדוגמה, גודל המשנה **int** ב-#C הוא 32 סיביות (bits). מכיוון של-**int** יש גודל קבוע, ניתן לדעת בדיוק את טווח הערכים שניתן להציב בו: 2147483648- עד 2147483647.

טיפ

כדי למצוא את הערך המקסימלי או המינימלי של משתנה **int** בקוד, ניתן להשתמש בשדות **Int32.MinValue** או **Int32.MaxValue**.

גודלו הקבוע של הסוג **int** יוצר בעיה. לדוגמה, מה יקרה אם תוסיף 1 למשתנה **Int** שערכו עומד על 2147483647? התשובה היא שהדבר תלוי באופן שבה התוכנית מהודרת. על פי ברירת המחדל, המהדר של #C יוצר קוד המאפשר לגלוש באופן חרישי (silently overflow).

במילים אחרות, אתה מקבל תשובה שאינה נכונה (למעשה, החישוב מדלג אל הערך השלילי השלם הגדול ביותר והתוצאה תהיה -2147483648.). הסיבה להתנהגות זו היא נוהל העיבוד: אריתמטיקה של מספרים שלמים היא פעולה שכיחה כמעט בכל תוכנית, והוספת מנגנון הבודק גלישה לכל ביטוי אריתמטי כזה יגרום לביצועים גרועים של התוכנית. במקרים רבים, הסיכון משתלם, כי אתה יודע (או מקווה) שהערכים מסוג int לא יגיעו בדרך כלל לטווח המקסימלי שלהם. אם גישה זו אינה נראית לך, תוכל להפעיל בדיקת גלישה (overflow checking) באמצעות ההגדרות.

טיפ

סביבת הפיתוח מאפשרת להפעיל ולכבות את בדיקת הגלישה באמצעות הגדרת מאפייני הפרויקט. בתפריט Project בחר Properties *YourProject* (במקום *YourProject* יופיע שם הפרויקט שלך). בתיבת הדו-שיח project properties, עבור לכרטיסייה Build ולחץ Advanced. בתיבת הדו-שיח Advanced Build Setting, סמן את תיבת הסימון Check for arithmetic overflow/underflow כדי להפעיל את בדיקת הגלישה.

אולם ללא תלות בבחירת אופן ההידור של התוכנה, תמיד תוכל להשתמש במילות המפתח **checked** ו-**unchecked** כדי להפעיל ולכבות באופן סלקטיבי את בדיקת הגלישה, בחלקים של היישום שלדעתך מחייבים בדיקה. מילות מפתח אלו עוקפות את הגדרות המהדר.

כתיבת משפטי checked

משפט checked הוא בלוק שמוצב אחרי מילת המפתח **checked**. כל האריתמטיקה העוסקת בשלמים בתוך המשפט checked תזרוק חריג מסוג **OverflowException** בכל פעם שחישוב של מספר שלם גולש (overflows), כפי שמוצג בדוגמה שלהלן:

```
int number = Int32.MaxValue;
checked
{
   int willThrow = number++;
   Console.WriteLine("this won't be reached");
}
```

חשוב

רק אריתמטיקת שלמים שנמצאת בתוך הבלוק תעבור בדיקה. לדוגמה, אם אחד המשפטים הנבדקים הוא קריאה לשיטה, הבדיקה לא תכלול את הפעולות המבוצעות במסגרת השיטה.

בנוסף, מילת המפתח **unchecked** יכולה לשמש ליצירת בלוקים שאינם נבדקים. כל האריתמטיקה בבלוקים כאלה לא תיבדק ולא ייזרק **OverflowException** במקרה של גלישה. לדוגמה:

```
int number = Int32.MaxValue;
unchecked
{
   int wontThrow = number++;
   Console.WriteLine("this will be reached");
}
```

חשוב

לא ניתן להשתמש במילות המפתח **checked** ו-**unchecked** כדי לשלוט בדיקה של אריתמטיקה של נקודה-צפה (floating point) או אריתמטיקה של מספרים שאינם שלמים. מילות המפתח **checked** ו-**unchecked** משפיעות על אריתמטיקת שלמים בלבד. אריתמטיקת נקודה-צפה לעולם לא תגרום לזריקה של **OverflowException**, אפילו לא במקרה של חלוקה ב-0.0 (ב- NET Framework. קיים ייצוג לאינסוף).

בדיקת ביטויים

ניתן להשתמש במילות המפתח **checked** ו-**unchecked** גם כדי לשלוט בבדיקת גלישה עבור ביטויים של מספרים שלמים, על ידי הוספת אחת ממילות המפתח לפני הסוגריים של הביטוי, כמודגם להלן:

```
int wontThrow = unchecked(Int32.MaxValue + 1);
int willThrow = checked(Int32.MaxValue + 1);
```

מילות המפתח **checked** ו-**unchecked** מאפשרות שליטה גם באופרטורים המורכבים (compound operators) (כגון =+ ו- =-) ובאופרטורים ++ ו- --. זכור כי x += y; זהה ל- x = x + y;.

בתרגיל הבא תלמד כיצד לבדוק אריתמטיקה בסביבת הפיתוח Visual Studio 2005.

בדוק ביטויים

1. חזור אל סביבת הפיתוח והצג את הפתרון MathsOperators.
2. בתפריט Debug בחר Start Without Debugging.
   כעת נסה לכפול שני ערכים גדולים.
3. הקלד 9876543 בתיבת הטקסט של האופרנד השמאלי, הקלד 9876543 בתיבת הטקסט של האופרנד הימני, תחת Operators סמן Multiplication ולחץ Calculate.
4. בתיבת הטקסט Result יופיע הערך 1195595903-. זהו ערך שלילי, ולפיכך בוודאות אינו נכון. ערך זה הוא תוצאת הכפל אשר גלשה מעבר לסף המקסימלי של **int**, מבלי להודיע למשתמש.
5. בחלונית Code המציגה את Form1.cs, אתר את השיטה MultiplyValues:

```
private int multiplyValues(int leftHandSide, int rightHandSide)
{
 expression.Text = leftHandSide.ToString() + " * " + rightHandSide.ToString();
 return leftHandSide * rightHandSide;
}
```

משפט ההחזרה (**return**) כולל את פעולת הכפל שגרמה לגלישה החרישית (silently overflow).

6. ערוך את משפט **ההחזרה** באופן שהערך המוחזר ייבדק. כעת, השיטה **MultiplyValues** נראית כך:

```
private int multiplyValues(int leftHandSide, int rightHandSide)
{
 expression.Text = leftHandSide.ToString() + " * " + rightHandSide.ToString();
 return checked(leftHandSide * rightHandSide);
}
```

פעולת הכפל תיבדק, ובמקום להחזיר תשובה שגויה, ייזרק חריג **verflowException**.

7. בחלונית Code, אתר את השיטה **calculate_Click**.

8. הוסף את המטפל **catch** הבא לאחר המטפל catch **FormatException** שבשיטה **calculate_Click**.

```
catch (OverflowException oEx)
{
   result.Text = oEx.Message;
}
```

טיפ

הגיון הפעולה של מטפל **catch** זה דומה לזה של המטפל **FormatException catch**. אולם כדאי בכל זאת להפריד בין שניהם ולא לכתוב מטפל **catch** כללי, מכיוון שייתכן שבשלב מאוחר יותר תחליט לטפל בשני החריגים הללו בצורות שונות.

9. בתפריט Debug בחר Start Without Debugging כדי לבנות ולהפעיל את היישום.

10. הקלד 9876543 בתיבת הטקסט של האופרנד השמאלי, הקלד 9876543 בתיבת הטקסט של האופרנד הימני, תחת Operators סמן Multiplication ולחץ Calculate.

המטפל **catch** השני יתפוס את החריג **OverflowException** ויציג את ההודעה "Arithmetic operation resulted in an overflow" בתיבת הטקסט Result.

11. לחץ Quit כדי לחזור לסביבת הפיתוח.

זריקת חריגים

נניח שברצונך להפעיל את השיטה **monthName** אשר מקבלת ארגומנט **int** יחיד ומחזירה את שם החודש המקביל אליו. לדוגמה, הקריאה **monthName(1)** תחזיר "January". השאלה היא, מה אמורה השיטה להחזיר כאשר הארגומנט השלם קטן מ-1 או גדול מ-12? התשובה הטובה ביותר לשאלה זו, שהיא לא תחזיר ערך אלא תזרוק חריג. ספריות המחלקה NET Framework. מכילות מחלקות חריגים רבות המיועדות במיוחד למצבים כמו זה. לרוב, אחת מהמחלקות תתאים למצב שעליך להתמודד איתו (אם לא, תוכל ליצור בקלות מחלקת חריגים משלך, אבל עליך ללמוד עוד על שפת #C לפני שתוכל לעשות זאת). במקרה שלנו, המחלקה הקיימת **ArgumentOutOfRangeException** מתאימה בדיוק לצרכינו:

```
public static string monthName(int month)
{
   switch (month)
   {
      case 1 :
         return “January”;
      case 2 :
         return “February”;
      ...
      case 12 :
         return “December”;
      default :
         throw new ArgumentOutOfRangeException(“Bad month”);
   }
}
```

שים לב כיצד מקרה (case) בשם **default** מפעיל את משפט **throw** כדי ליצור חריג. המשפט **throw** זקוק לחריג שיוכל לזרוק. דוגמה זו מנצלת ביטוי שיוצר אובייקט **ArgumentOutOfRangeException** חדש. האובייקט מאותחל עם מחרוזת שמוצבת במאפיין **Message** שלו על ידי בנאי. על **בנאים** (constructors) נלמד בהרחבה בפרק 7.

בתרגיל הבא עליך להוסיף קוד אשר יזרוק ויתפוס חריגים בפרויקט MathsOperators.

זרוק חריגים משלך

1. חזור אל סביבת הפיתוח. ודא שהפתרון MathsOperators עודנו פתוח.
2. בתפריט Debug בחר Start Without Debugging.
3. הקלד **24** בתיבת הטקסט של האופרנד השמאלי, הקלד **36** בתיבת הטקסט של האופרנד הימני ולחץ Calculate.

   הערך **0** יופיע בתיבת הטקסט Result. העובדה שלא בחרת באחד מהאופרטורים, לא משפיעה באופן ברור על התנהלות התוכנית. במקרה זה רצוי שתופיע הודעת אבחון בתיבת הטקסט Result.
4. לחץ Quit כדי לחזור לסביבת הפיתוח.
5. בחלונית Code המציגה את Form1.cs, אתר את השיטה **doCalculation**. השיטה נראית כך:

```
private int doCalculation(int leftHandSide, int rightHandSide)
{
   int res = 0;
   if (addition.Checked)
      res = addValues(leftHandSide, rightHandSide);
   else if (subtraction.Checked)
      res = subtractValues(leftHandSide, rightHandSide);
```

```
   else if (multiplication.Checked)
      res = multiplyValues(leftHandSide, rightHandSide);
   else if (division.Checked)
      res = divideValues(leftHandSide, rightHandSide);
   else if (remainder.Checked)
      res = remainderValues(leftHandSide, rightHandSide);
   return res;
}
```

השדות **addition**, **subtraction**, **multiplication**, **division** ו-**remainder** הם לחצני אפשרויות שמופיעים בקבוצה Operators שבטופס. לכל לחצן אפשרויות יש מאפיין בוליאני בדוק (**checked**), המכיל את הערך true כאשר הלחצן מסומן. המשפט **if** המדורג בודק את לחצני האפשרויות בזה אחר זה כדי למצוא איזה מביניהם מסומן. אם אין לחצן מסומן, כל משפטי **if** יהיו false ולכן לא יבוצעו, והמשתנה **res**, אשר מכיל את הערך המוחזר על ידי השיטה,ישמור על ערכו ההתחלתי (0).

חשוב

אל תתבלבל בין מילת המפתח **checked** ובין המאפיין **Checked** של לחצן האפשרויות - הם אינם קשורים זה לזה בשום אופן שהוא.

ניתן לנסות לפתור בעיה זו על ידי הוספת משפט else נוסף למערך הקינון if-else, אשר ידפיס הודעה בתיבת הטקסט result, באופן הבא:

```
if (addition.Checked)
   res = addValues(leftHandSide, rightHandSide);
...
else if (remainder.Checked)
   res = remainderValues(leftHandSide, rightHandSide);
else
   result.Text = “no operator selected”;
```

פתרון זה אינו אידיאלי, כי מטרת השיטה אינה להדפיס הודעות. קוד זה יגרום לכך שהתוכנית תכיל שתי שיטות אשר מדפיסות הודעות אבחון בתיבת הטקסט **result**: **calculate_Click** ו-**doCalculation**. לכן, עדיף להפריד בין זיהוי שגיאות והכרזה עליהן, לבין זריקת שגיאות וטיפול בהן.

.6 הוסף משפט **else** לרשימת המשפטים **if-else** (ממש לפני משפט ההחזרה **return**) וזרוק חריג **InvalidOperationException** באופן הבא:

```
else
   throw new InvalidOperationException(“no operator selected”);
```

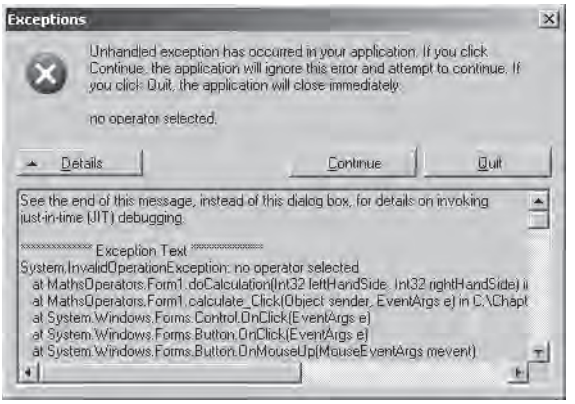
.7 בתפריט Debug בחר Start Without Debugging כדי לבנות ולהפעיל את היישום.

.8 הקלד **24** בתיבת הטקסט של האופרנד השמאלי, הקלד **36** בתיבת הטקסט של האופרנד הימני ולחץ Calculate.

על המסך תוצג ההודעה Exceptions.

9. לחץ Details.

ההודעה תציין שהחריג **InvalidOperationException** נזרק כשהוא מכיל את המחרוזת "no operator selected".



10. בתיבת ההודעה Exceptions לחץ Quit.

היישום יסתיים ותחזור לסביבת הפיתוח.

לאחר כתיבת משפט **throw** ובדיקה שהוא אכן זורק חריג, עליך לכתוב מטפל **catch** אשר יתפוס את החריג הזה.

כתוב חריג משלך

1. בחלונית Code המציגה את Form1.cs, אתר את השיטה **calculate_Click**.

2. הוסף את המטפל catch הבא מייד לאחר שני המטפלים שכבר נמצאים בשיטה **calculate_Click**:

```
catch (InvalidOperationException ioEx)
{
   result.Text = ioEx.Message;
}
```

קוד זה יתפוס את החריג **InvalidOperationException** שנזרק, אם לא בחרת באחד מהאופרטורים.

3. בתפריט Debug בחר Start Without Debugging.

4. הקלד **24** בתיבת הטקסט של האופרנד השמאלי, הקלד **36** בתיבת הטקסט של האופרנד הימני ולחץ Calculate.

בתיבת הטקסט Result תוצג ההודעה "no operator selected".

5. לחץ Quit.

כעת היישום יציב הרבה יותר משהיה קודם לכן. אולם, יש עדיין מספר חריגים שעשויים להתעורר מבלי להיתפס, ואשר עשויים למוטט את היישום. לדוגמה, אם תנסה לחלק באפס, ייזרק חריג **DividedByZeroException**. אחד הפתרונות לבעיה יהיה כתיבת מספר גדול יותר של מטפלי **catch** בשיטה **calculate_Click**. אולם פתרון עדיף הוא הוספה של מטפל **catch** כללי בסוף רשימת המטפלים, שתפקידו לתפוס את Exception. כך ניתן לתפוס את כל החריגים שאינם מטופלים לפניו.

טיפ

ההחלטה האם לתפוס את **כל** החריגים הבלתי מטופלים בשיטה, תלויה בסוג היישום שאתה בונה. במקרים מסוימים, עדיף לתפוס את החריגים כמה שיותר קרוב לנקודה שהם "נוצרים" בה. במקרים אחרים עדיף לחכות שהחריג יועבר במעלה הרמות של הקריאות, כדי שיטופל על ידי השיטה אשר קראה לשגרה (routine) שבה נוצר.

6. בחלונית Code המציגה את Form1.cs, אתר את השיטה **calculate_Click**.

7. הוסף את המטפל **catch** הבא בסוף רשימת מטפלי **catch** הקיימים:

```
catch (Exception ex)
{
   result.Text = ex.Message;
}
```

מטפל זה יתפוס את כל החריגים שלא טופלו עד כה, ללא תלות בסוג.

8. בתפריט Debug בחר Start Without Debugging.

כעת בצע מספר חישובים כדי ליצור במתכוון חריגים, כדי לוודא שכולם אכן נתפסים.

9. הקלד **24** בתיבת הטקסט של האופרנד השמאלי, הקלד **36** בתיבת הטקסט של האופרנד הימני ולחץ Calculate.

ודא את הופעת ההודעה "no operator selected" בתיבת הטקסט Result. הודעה זו נוצרה על ידי המטפל **InvalidOperationException**.

10. בתיבת הטקסט של האופרנד השמאלי הקלד **John** ולחץ Calculate. ודא את הופעת ההודעה "Input string was not in correct format" בתיבת הטקסט Result. הודעה זו נוצרה על ידי המטפל **FormatException**.

11. הקלד **24** בתיבת הטקסט של האופרנד השמאלי, הקלד **0** בתיבת הטקסט של האופרנד הימני, תחת Operators סמן Divide ולחץ Calculate. ודא את הופעת ההודעה "Attempted to divide by zero" בתיבת הטקסט Result. הודעה זו נוצרה על ידי המטפל **Exception**.

12. לחץ Quit.

שימוש בבלוק finally

חשוב לזכור שבעת זריקת חריג, זרימת התוכנית (flow of execution) משתנה. משמעות הדבר היא שאינך יכול להבטיח שמשפט תמיד יופעל בסיום המשפט שלפניו, כי המשפט שלפניו עשוי לזרוק חריג. הבט בדוגמה הבאה. קל להניח שהקריאה לשיטה **reader.Close** תפעל תמיד, כי הרי היא נמצאת ברצף קוד:

```
TextReader reader = src.OpenText();
string line;
while ((line = reader.ReadLine()) != null)
{
   source.Text += line + "\n";
}
reader.Close();
```

לעיתים אין בעיה שמשפט אחד לא יופעל, אולם במקרים רבים זו עשויה להיות תקלה חמורה. אם התוכנית משחררת משאב אשר נרכש במשפט קודם, אזי אי-הפעלת המשפט תמנע את שחרור המשאב. דוגמה זו ממחישה מצב כזה: אם הקריאה ל-**src.OpenText** מבוצעת, יוקצה משאב (מטפל בקובץ - a file handle) ועליך לוודא קריאה לשיטה **reader.Close** כדי לשחרר אותו. אם לא, במוקדם או במאוחר יאזלו המטפלים בקבצים, ולא תוכל לפתוח קבצים נוספים. אם מטפלים בקבצים טריוויאליים מדי עבורך, חשוב במקום זה על קישור לבסיס נתונים.

כדי להבטיח שמשפט כלשהו יופעל תמיד, גם במקרה שנזרק חריג, יש לכתוב את המשפט בתוך בלוק **finally**. בלוק **finally** מופעל מייד לאחר בלוק **try**, או לאחר מטפל **catch** האחרון שאחרי הבלוק **try**. כל עוד התוכנית תיכנס לבלוק **try** הקשור לבלוק **finally**, הבלוק **finally** **יופעל תמיד**, גם במקרה של היווצרות חריג. אם חריג נזרק ונתפס במסגרת מקומית, יופעל תחילה המטפל של החריג, ולאחריו הבלוק **finally**. אם החריג נתפס מחוץ למסגרת המקומית (כלומר שסביבת ההרצה (CLR) צריכה הייתה לסרוק במעלה רשימת הקריאות לשיטות כדי למצוא מטפל מתאים), הבלוק **finally** יופעל תחילה. בכל מקרה, הבלוק **finally** יופעל בשלב זה או אחר.

הפתרון לבעיית reader.Close הוא:

```
TextReader reader = null;
try
{
   reader = src.OpenText();
   string line;
   while ((line = reader.ReadLine()) != null)
   {
      source.Text += line + "\n";
   }
}
finally
{
```

```
    if (reader != null)
    {
        reader.Close();
    }
}
```

גם אם נזרק חריג, הבלוק **finally** מבטיח שהמשפט **reader.Close** יופעל תמיד. בפרק 13 נציג דרך נוספת לפתור קושי זה.

☯ **אם ברצונך להמשיך לפרק הבא**

השאר את Visual Studio 2005 פעילה ועבור לפרק 7.

☯ **אם ברצונך לסגור את Visual Studio 2005**

בתפריט File בחר Exit. אם מוצגת תיבת דו-שיח Save, לחץ Yes.

פרק 6 – טבלה מסכמת

המשימה	צריך
לזרוק חריג.	להשתמש במשפט **throw**. לדוגמה: `throw new FormatException(source);`

לוודא שאריתמטיקה של מספרים שלמים תמיד תיבדק למציאת גלישה.	להשתמש מילת המפתח **check**. לדוגמה:
	`int number = Int32.MaxValue;`
	`checked`
	`{`
	`  number++;`
	`}`

לתפוס חריג מסוים.	לכתוב מטפל **catch** אשר תופס חריג מסוים:
	`try`
	`{`
	`  ...`
	`}`
	`catch (FormatException fEx)`
	`{`
	`  ...`
	`}`

לתפוס את כל החריגים באמצעות מטפל אחד.	לכתוב מטפל **catch** אשר תופס את כל סוגי החריגים. לדוגמה:
	`try`
	`{`
	`  ...`
	`}`
	`catch (Exception Ex)`
	`{`
	`  ...`
	`}`

לוודא שקוד מסוים יפעל תמיד, גם בעת זריקת חריג.	לכתוב את הקוד בתוך בלוק **finally**. לדוגמה:
	`try`
	`{`
	`  ...`
	`}`
	`finally`
	`{`
	`  // always run`
	`}`

חלק ב'

הבנת שפת #C

בחלק זה:

פרק 7

יצירה וניהול של מחלקות ואובייקטים

בסיום פרק זה, תוכל:

- להגדיר **מחלקה** (class) המכילה קבוצה של שיטות ופרטי נתונים בעלי מכנה משותף.
- לשלוט בנגישות של חברי המחלקה באמצעות מילות המפתח public ו-private.
- ליצור **אובייקטים** (objects) באמצעות מילת המפתח **new** ובנאי.
- לכתוב **בנאים** (constructors) משלך ולקרוא להם.
- ליצור שיטות ונתונים אשר נגישים לשימושם של כל המופעים של אותה מחלקה, באמצעות מילת המפתח **static**.

בחלק א' של הספר למדת להכריז על משתנים, ליצור ערכים באמצעות אופרטורים (operators), לקרוא לשיטות (methods) ולכתוב חלק גדול מהמשפטים (statements) הדרושים ליישום של שיטה. כעת ניתן להתקדם לשלב הבא - שילוב של שיטות ונתונים למחלקות (classes) משלך.

טכנולוגיית Microsoft .NET Framework מכילה אלפי מחלקות, שבחלקן כבר השתמשת קודם, וביניהן **Console** ו-**Exception**. המחלקות מספקות מנגנון נוח לעיצוב ישויות שנשלטות על ידי יישומים. ישות שכזו עשויה לייצג פריט מסוים, כגון לקוח, או משהו מופשט יותר, כגון עסקה פיננסית. חלק מתהליך התכנון של כל מערכת עוסק באיתור הישויות החשובות, ולאחר מכן שלב ניתוח שמטרתו לקבוע איזה מידע עליהן להכיל ואילו פונקציות עליהן לבצע. את המידע שמכילה המחלקה עליך לשמור בשדות (fields), וליישם את הפונקציות של המחלקה באמצעות שיטות.

הפרקים בחלק ב' מכילים את כל אשר עליך לדעת כדי ליצור מחלקות משלך.

הבנת הסיווג

בין הסיווג (classification) לבין המחלקה קיים קשר חזק. בעת עיצוב מחלקה, הינך מארגן את המידע באופן שיטתי בישות בעלת משמעות. ארגון זה הינו למעשה סיווג, ופעולה זו מבוצעת באופן יומיומי, לא רק על ידי מתכנתים. לדוגמה, לכל המכוניות מספר מכנים משותפים (היגוי, בלימה, האצה וכו'). המילה מכונית משמשת להתייחסות אל העצמים אשר כוללים את התכונות והמאפיינים המשותפים האלה. כל עוד יש הסכמה כוללת לגבי משמעות המילה, הכל יהיה ברור לכולם. כך ניתן להביע באופן תמציתי רעיונות מורכבים ומדויקים. בהיעדר סיווג, קשה לדמיין כיצד היו יכולים אנשים לחשוב ולתקשר.

לאור העובדה שהסיווג מוטמע בדרכי החשיבה והתקשורת שלנו, טבעי הדבר שבעת כתיבת תוכניות נסווג למחלקות את הרעיונות השונים הנובעים מבעיה ופתרונה, ולאחר מכן נעצב מחלקות אלו על פי שפת התכנות. זהו בדיוק התהליך אשר שפות תכנות מונחה עצמים מתקדמות, כגון Microsoft Visual C#, מאפשרות למשתמש.

מטרת הכימוס

כימוס (encapsulation) הינו עיקרון חשוב בעת הגדרת מחלקות. הרעיון הוא שתוכנית אשר משתמשת במחלקה לא אמורה לטפל במנגנוני הפעולה הפנימיים שלה. התוכנית יוצרת **מופעים** (instances) של המחלקה, ואז מפעילה **שיטות** (methods) של אותה מחלקה. כל עוד שיטות אלו מבצעות את יעודן המוגדר, התוכנית אינה צריכה לעסוק בדרך הביצוע שלהן. לדוגמה, כאשר הינך קורא לשיטה **Console.WriteLine**, אין לך עניין כיצד המחלקה Console מארגנת את הנתונים אשר יוצגו במסך. ייתכן מאוד שהמחלקה משתמשת במידע פנימי רב כדי לבצע את השיטות השונות הכלולות בה. המידע והפעולות הנוספות אינם נחשפים בפני התוכנית שמפעילה את המחלקה. לפיכך, כימוס מכונה לעיתים **הסתרת-מידע** (information hiding). לכימוס יש למעשה שתי מטרות:

1. לשלב שיטות ונתונים במחלקה, או במילים אחרות, לתמוך בסיווג.
2. לשלוט בנגישות לשיטות ולנתונים, או במילים אחרות, לשלוט בצורת הפעלת המחלקה.

הגדרת מחלקה ושימוש בה

כדי להגדיר מחלקה חדשה ב-C# עליך להשתמש במילת המפתח **class**, בשם המחלקה ובצמד סוגריים מסולסלים. הנתונים והשיטות של המחלקה נמצאים בגוף המחלקה, בין הסוגריים המסולסלים. להלן דוגמה של מחלקה בשם **Circle** המכילה שיטה אחת לחישוב שטח עיגול, וערך אחד, שהינו רדיוס העיגול.

```
class Circle
{
    double Area()
    {
        return 3.141592 * radius * radius;
    }

    double radius;
}
```

גוף המחלקה מכיל, אם כן, שיטות רגילות (כגון **Area**) ושדות (כגון **radius**). זכור שהמשתנים במחלקה תמיד מכונים **שדות** (fields). כבר למדת כיצד להכריז על **משתנים** בפרק 2, וכיצד לכתוב שיטות בפרק 3, ולמעשה כמעט ולא קיים כאן רכיב תחביר (syntax) שאינך מכיר.

השימוש במחלקה **Circle** דומה לשימוש בסוגים אחרים שכבר פגשת. אתה יוצר משתנה מסוג **Circle**, ולאחר מכן מאתחל אותו באמצעות נתון תקני. להלן דוגמה:

```
Circle c;// Create a Circle variable
c = new Circle();// Initialize it
```

שים לב לשימוש במילת המפתח new. קודם לכן, כאשר אתחלת משתנה מסוג **int** או **float**, קבעת לו ערך:

```
int i;
i = 42;
```

לא ניתן לעשות זאת עם משתני המחלקה השונים. אחת הסיבות לכך היא שב- C# אין סוג תחביר המאפשר להציב ערכי מחלקה ליטרליים (literal class values) במשתנים (מהו ה-**Circle** המקביל ל-42?). סיבה נוספת קשורה לאופן שבו מוקצה ומנוהל הזיכרון עבור משתני המחלקה על ידי סביבת ההרצה CLR (Common Language Runtime). נדון בנושא זה בפרק 8. בשלב זה, קבל את העובדה שמילת המפתח new יוצרת מופע מחלקה חדש, שנקרא לעיתים תכופות אובייקט (Object).

חשוב

הפרד בין המושגים **מחלקה** לבין **אובייקט**. מחלקה הינה ההגדרה של **סוג** (type). אובייקט הינו **מופע** (instance) של סוג זה, הנוצר בזמן הרצת התוכנית. לדוגמה, ניתן ליצור מופעים רבים של המחלקה **Circle** בתוכנית כלשהי על ידי שימוש במילת המפתח **new**, כפי שניתן ליצור בתוכנית משתני **int** רבים. כל אחד מהמופעים של המחלקה **Circle** הינו למעשה אובייקט התופס שטח משלו בזיכרון, ופועל בצורה עצמאית מכל שאר המקרים.

שליטה בנגישות

באופן מפתיע, למחלקה **Circle** אין עדיין כל שימוש מעשי. כאשר אתה **מכַמֵּס** (encapsulate) את השיטות והנתונים בתוך מחלקה, נוצר חוצץ בין המחלקה לבין העולם החיצון. שדות (כגון **radius**) ושיטות (כגון **Area**) המוגדרים במסגרת המחלקה, גלויים לשיטות אחרות במחלקה, אולם אינם גלויים לעולם החיצון. הם הופכים ל"רכוש" הפרטי של המחלקה. במילים אחרות, למרות שבאפשרותך ליצור אובייקט Circle בתוכנית, אין לך גישה לשדה **radius** ואינך יכול לקרוא לשיטה **Area**, וזאת למעשה הכוונה באמירה שלמחלקה אין כל שימוש מעשי - עדיין! אולם, באפשרותך לשנות את הגדרות שדה או השיטה באמצעות מילות המפתח **public** ו-**private**, ובכך לשלוט בנגישות מבחוץ:

- שיטה או שדה נחשבים **לפרטיים** (**private**) כאשר הם נגישים רק מתוך המחלקה. כדי להכריז על שיטה או שדה כפרטיים, עליך להקליד את מילת המפתח **private** לפני הכרזתם. אפשרות זו היא ברירת המחדל, אולם רצוי בכל זאת לציין במפורש שהשיטה או השדה פרטיים, כדי למנוע בלבול.
- שיטה או שדה נחשבים **לציבוריים** (**public**) כאשר הגישה אליהם מתאפשרת מתוך ומחוץ למחלקה כאחד. כדי להכריז על שיטה או שדה כציבוריים, עליך להקליד את מילת המפתח **public** לפני הכרזתם.

להלן המחלקה **Circle** פעם נוספת, אולם הפעם השיטה **Area** מוגדרת כציבורית בעוד שהשדה radius מוגדר כפרטי:

```
class Circle
{
    public double Area()
    {
        return 3.141592 * radius * radius;
    }

    private double radius;
}
```

הערה

מתכנתי ++C צריכים לשים לב שאין נקודתיים לאחר מילת המפתח **private** או **public**. יש לחזור על מילת המפתח בכל הכרזה.

שים לב ש-**radius** מוכרז כשדה פרטי ואין גישה אליו מחוץ למחלקה, אך יש גישה אליו מתוך המחלקה **Circle**. לכן, מכיוון שהשיטה **Area** נמצאת בתוך המחלקה, יש גישה לשדה **radius**. ערך המחלקה עודנו מוגבל, כי אין דרך לאתחל את השדה **radius**. כדי לתקן זאת עלינו להיעזר **בבנאי** (constructor).

טיפ

השדות במחלקה מאותחלים באופן אוטומטי ל-**0**, **false** או **null**, בהתאם לסוגם. עם זאת, רצוי ליצור מנגנון מסודר לאתחול המשתנים.

חשוב

אל תכריז על שני חברי מחלקה **ציבוריים** אשר הדבר היחיד המבדיל ביניהם הוא אותיות רישיות או רגילות (כמו A לעומת a). אחרת, המחלקה לא תעמוד ב"דרישות השפה המשותפת" - CLS, ולא תוכל לשמש שפות תכנות אחרות אשר אינן מבחינות בהבדלים אלה, כגון Microsoft Visual Basic.

עבודה עם בנאים

בעת שימוש במילת המפתח **new** ליצירת אובייקט, סביבת ההרצה (CLR) צריכה לבנות את האובייקט באמצעות הגדרות המחלקה. סביבת ההרצה צריכה לגזול חלק מהזיכרון של מערכת ההפעלה, למלאו בשדות אשר הוגדרו על ידי המחלקה, ואז לקרוא לבנאי לבצע את האתחולים הנחוצים.

בנאי (**constructor**) הוא שיטה מיוחדת ששמה זהה לשם המחלקה. הוא מסוגל לקבל פרמטרים, אולם אינו יכול להחזיר ערך (אפילו לא void). בכל מחלקה חייב להיות בנאי. אם לא תכתוב אחד בעצמך, המהדר ייצור עבורך בנאי ברירת מחדל באופן אוטומטי, אך זהו בנאי שלד בלבד שאינו עושה דבר. באפשרותך לכתוב בנאי ברירת מחדל בקלות: עליך רק להוסיף שיטה ציבורית בעלת שם זהה לזה של המחלקה, ואשר אינה מחזירה ערך. בדוגמה שלהלן ניתן לראות את המחלקה **Circle**, הכוללת בנאי ברירת מחדל המאתחל את השדה **radius** לאפס:

```
public Circle() // default constructor
{
   radius = 0.0;
}
```

הערה

מבחינת #C, המשמעות של בנאי **ברירת המחדל** היא שהוא אינו מקבל פרמטרים. אין חשיבות אם הבנאי נוצר על ידי המהדר או אם כתבת אותו בעצמך; הוא נחשב "בנאי ברירת מחדל". ניתן ליצור בנאים שאינם ברירת מחדל, כפי שתלמד בהמשך פרק זה, בסעיף "העמסת בנאים".

שים לב שהבנאי מסומן כציבורי. אם מילת המפתח הזו מושמטת, הבנאי נחשב פרטי (בדומה לשיטות ולשדות). כאשר הבנאי פרטי, לא ניתן לעשות בו שימוש מחוץ למחלקה, והדבר ימנע יצירת אובייקטים של **Circle** על ידי שיטות שאינן שייכות למחלקה **Circle**. אין להסיק מכך שבנאים פרטיים הינם חסרי ערך. יש להם מספר שימושים, אך לא נדון בהם כעת.

בשלב זה נשתמש במחלקה **Circle** ונפעיל את השיטה **Area** המשתייכת אליה. שים לב לשימוש בסימון נקודה (dot notation) כדי להפעיל את השיטה **Area** על האובייקט **Circle**:

```
Circle c;
c = new Circle();
double areaOfCircle = c.Area();
```

העמסת בנאים

כמעט סיימנו, אבל רק כמעט. ביכולתך להכריז על משתנה **Circle**, להציב בו הפניה לאובייקט **Circle** אשר זה עתה נוצר, ולבסוף לקרוא לשיטה **Area** השייכת לו. אולם, נותרה בעיה אחת: השטח של כל האובייקטים **Circle** יהיה תמיד 0 כי **בנאי ברירת המחדל** (default constructor) מאתחל את הרדיוס ל-0, וכך הוא גם נשאר, כי השדה פרטי ואין כל דרך לשנות את ערכו לאחר האתחול. אחת הדרכים לפתרון בעיה זו היא להבין שהבנאי הוא למעשה סוג מיוחד של שיטה, ובדומה לשאר השיטות, ניתן להעמיסו (overload), ומכאן המושג **העמסת בנאים** (Overloading Constructors). כפי שקיימות מספר גרסאות לשיטה **Console.WriteLine**, אשר כל אחת מהן מקבלת פרמטרים אחרים, כך גם ניתן לכתוב גרסאות שונות של בנאי. באפשרותך להוסיף למחלקה **Circle** בנאי אשר מקבל את הרדיוס בתור פרמטר. הנה כך:

```
class Circle
{
   public Circle() // default constructor
   {
      radius = 0.0;
   }
   public Circle(double initialRadius) // overloaded constructor
   {
      radius = initialRadius;
   }
   public double Area()
   {
      return 3.141593 * radius * radius;
   }
   private double radius;
}
```

הערה

סדר הבנאים במחלקה אינו חשוב. סדר אותם כפי שנוח לך.

יש בעיה ב-C# שכדאי להיות מודעים אליה: אם תכתוב בעצמך בנאי עבור המחלקה, המהדר לא יצור עבורך בנאי ברירת מחדל. אי-לכך, אם כתבת בנאי אשר מקבל פרמטר אחד או יותר ואתה עדיין רוצה בבנאי ברירת מחדל, תצטרך לכתוב אותו בעצמך.

מחלקות חלקיות

מחלקה עשויה להכיל מספר שיטות, שדות, בנאים וגם מספר רכיבים שנלמד בהמשך. מחלקה מתפקדת עשויה להיות גדולה למדי. תכונה חדשה ב- C# 2.0 מאפשרת לפצל את קוד המקור של המחלקה לקבצים נפרדים, כדי שאפשר יהיה לארגן את ההגדרות של מחלקה גדולה בחלקים קטנים ונוחים יותר לשליטה. תכונה זו משמשת את יישומי סביבת העבודה Visual Studio 2005 for Windows Forms, שקוד המקור בהם ניתן לעריכה על ידי מפתח היישום (developer). הקוד שכתב כל מפתח נשמר בקובץ נפרד מהקוד המופק על ידי סביבת הפיתוח בכל פעם שתצורת הטופס (form) משתנה.

בעת פיצול מחלקה למספר קבצים, עליך להגדיר את חלקי המחלקה באמצעות מילת המפתח **partial** (חלקי) בכל אחד מהקבצים. לדוגמה, אם המחלקה **Circle** פוצלה בין שני קבצים: הקובץ circ1.cs שמכיל את הבנאים, והקובץ circ2.cs שמכיל את השיטות והשדות, התוכן של קובץ circ1.cs יהיה:

```
partial class Circle
{
   public Circle() // default constructor
   {
      this.radius = 0.0;
   }
   public Circle(double initialRadius) // overloaded constructor
   {
      this.radius = initialRadius;
   }
}
```

התוכן של circ2.cs יהיה:

```
partial class Circle
{
   public double Area()
   {
      return Math.PI * radius * radius;
   }
   private double radius;
}
```

בזמן ההידור של מחלקה אשר פוצלה לקבצים נפרדים, עליך לספק למהדר את כל הקבצים.

בתרגיל שלהלן עליך להכריז על מחלקה בעלת שני בנאים ציבוריים ושני שדות פרטיים. עליך ליצור מופעים של המחלקה באמצעות מילת המפתח **new** וקריאה לבנאים.

כתוב בנאים וצור אובייקטים

1. הפעל את Microsoft Visual Studio 2005.
2. פתח את הפרויקט Classes, שבתיקייה My Documents\Microsoft Press\Visual CSharp Step by Step\Chapter 7\Classess.
3. הצג את הקובץ Program.cs בחלון Code and Text Editor, ואתר את השיטה Main של המחלקה Program.

   השיטה **Main** קוראת לשיטה **Entrance**, שעטופה בבלוק **try** שלאחריו מטפל **catch**. בלוק **try/catch** זה מאפשר לרשום קוד אשר במקרים רגילים ייכלל תחת **Main** בשיטה **Entrance**, תוך ביטחון שיהיה ניתן להבחין בכל חריגה.
4. הצג את הקובץ Point.cs בחלון Code and Text Editor.

   המחלקה **Point** עודנה ריקה. היא איננה מכילה בנאים ולפיכך המהדר יכתוב עבורך בנאי ברירת מחדל. כעת עליך להפעיל את הבנאי שנבנה על ידי המהדר.
5. חזור לקובץ Program.cs ומצא את השיטה **Entrance** אשר במחלקה **Program**. ערוך את גוף השיטה **Entrance** כדי שתכלול את המשפט הבא:

```
Point origin = new Point();
```

6. בתפריט Build לחץ Build Solution.

   הקוד נבנה ללא שגיאה, כי המהדר יוצר את הקוד עבור בנאי ברירת מחדל של המחלקה **Point**. עם זאת, אינך יכול לראות את הקוד של C# עבור הבנאי הזה כי המהדר אינו יוצר משפטים בשפת המקור (source language).
7. חזור למחלקה **Point** בקובץ Point.cs. הוסף בנאי **ציבורי** אשר מקבל שני ארגומנטים מסוג int וקורא ל-**Console.WriteLine** כדי להציג את ערכי הארגומנטים במסך.

   המחלקה **Point** נראית כך:

```
class Point
{
   public Point(int x, int y)
   {
      Console.WriteLine("x:{0}, y:{1}", x, y);
   }
}
```

הערה

השיטה **Console.WriteLine** משתמשת בערכים **{0}** ו-**{1}** כשומרי מקום (placeholders). כאשר התוכנית תופעל, במשפט שלעיל **{0}** יוחלף בערך של **x**, בעוד **{1}** יוחלף בערך של y.

8. בתפריט Build לחץ Build Solution.

המהדר ידווח כעת על שגיאה:

```
'Classes.Point' does not contain a constructor that takes '0' arguments
```

הקריאה לבנאי ברירת המחדל בשיטה **Entrance** אינה פועלת יותר, כי בנאי ברירת המחדל אינו קיים עוד. מכיוון שיצרת בנאי משלך עבור המחלקה **Point**, המהדר לא יצר בנאי ברירת מחדל באופן אוטומטי. עליך לתקן זאת על ידי יצירת בנאי ברירת מחדל בעצמך.

9. ערוך את המחלקה **Point** והוסף בנאי ברירת מחדל **ציבורי** הקורא ל-**Console.WriteLine** כדי לכתוב את המחרוזת **"default constructor called"** על המסך.

המחלקה **Point** נראית כך:

```
class Point
{
   public Point()
   {
      Console.WriteLine("default constructor called");
   }
   public Point(int x, int y)
   {
      Console.WriteLine("x:{0}, y:{1}", x, y);
   }
}
```

10. בקובץ Program.cs, ערוך את גוף השיטה **Entrance**. הכרז על משתנה **bottomRight** מסוג **Point**, ואתחל אותו לאובייקט **Point** חדש באמצעות בנאי המקבל שני ארגומנטים: 1024 ו-1280.

השיטה **Entrance** נראית כך:

```
static void Entrance()
{
    Point origin = new Point();
    Point bottomRight = new Point(1024, 1280);
}
```

11. בתפריט Debug לחץ Start Without Debugging.

הקוד נבנה כעת ללא שגיאות, ופועל. ההודעות הבאות יופיעו במסך:

```
default constructor called
x:1024, y:1280
```

12. הקש Enter.

חלון התצוגה נסגר, ושוב תופיע על המסך סביבת הפיתוח. כעת עליך להוסיף שני שדות מסוג int למחלקה **Point** ולשנות את הבנאים כדי שיאתחלו את השדות האלה.

13. ערוך את המחלקה Point והוסף שני שדות מופע פרטיים (private instance fields) מסוג int הנקראים x ו–y.

המחלקה **Point** נראית כך:

```
class Point
{
   public Point()
   {
      Console.WriteLine("default constructor called");
   }

   public Point(int x, int y)
   {
      Console.WriteLine("x:{0}, y:{1}", x, y);
   }

      private int x, y;
}
```

כעת עליך לערוך את הבנאי השני של **Point** כדי שיאתחל את השדות **x** ו–**y** לפי הערכים של הפרמטרים **x** ו–**y**. בתהליך זה טמונה מלכודת פוטנציאלית. אם לא תנקוט בזהירות, הבנאי ייראה כך:

```
public Point(int x, int y) // Don't type this in!
{
   x = x;
   y = y;
}
```

קוד זה יעבור את שלב ההידור, אך נראה שיש סתירה במשפטים אלה. כיצד יכול המהדר להבין מהמשפט x = x; שה–x הראשון הוא השדה וה–x השני הוא הפרמטר? לא ניתן להבין זאת! למעשה, כל בנאי מציב את הפרמטרים בעצמו, והוא אינו משנה את השדות כלל. אין ספק שזו אינה התוצאה המתבקשת.

הפתרון לבעיה זו הוא שימוש במילת המפתח **this** (זה), כדי להפריד בין המשתנים שהם פרמטרים לבין אלה שהם שדות. הוספת **this** לפני המשתנה אומרת: "זהו השדה באובייקט זה".

14. שנה את הבנאי Point באופן הבא:

```
public Point(int x, int y)
{
   this.x = x;
   this.y = y;
}
```

15. ערוך את בנאי ברירת המחדל **Point** כדי שיאתחל את השדות **x** ו–**y** לערך 1–, והסר את המשפט **Console.WriteLine**. בנוסף, למרות שאין פרמטרים העשויים ליצור בלבול, מומלץ לציין באמצעות מילת המפתח **this** מהו השדה. הנה כך:

```
public Point()
{
   this.x = -1;
   this.y = -1;
}
```

16. בתפריט Build לחץ Build Solution. ודא שהקוד מהודר ללא שגיאות או אזהרות. תוכל גם להפעילו, אולם הוא עדיין לא יניב פלט.

קביעת שמות ונגישותם

המלצות Net Framework. הבאות מתייחסות לדרך שבה מקובל לקבוע שמות לשדות ולשיטות, בהתאם לנגישותם של חברי המחלקה:

- **מזהים** (identifiers) **ציבוריים** צריכים להתחיל באות רישית. לדוגמה, **Area** נכתבת באות A (ולא a), כי הכוונה **לציבורי**. מערכת כללים זו מוכרת בשם **PascalCase** naming scheme, כי שימשה לראשונה בשפת פסקל.
- מזהים שאינם **ציבוריים**, ובכלל זה משתנים מקומיים, צריכים להתחיל באות רגילה. לדוגמה, **radius** נכתבת באות r (ולא R), כי הכוונה **לפרטי**. שיטת סימון זו מוכרת בשם **camelCase** naming scheme.

לחוק זה יוצא מן הכלל אחד בלבד: שמות של מחלקות מתחילים באות רישית ושמות של בנאים צריכים להיות זהים בדיוק לאלה של המחלקות שהם שייכים להן. לכן, גם בנאי **פרטי** יתחיל באות רישית.

שיטות השייכות למחלקה ופועלות על בסיס מידע השייך למופע מסוים של המחלקה, הן **שיטות מופע** (**instance methods**). בתרגיל הבא עליך לכתוב שיטת מופע בשם **DistanceTo** עבור המחלקה **Point**, אשר מחשבת את המרחק בין שתי נקודות.

כתוב שיטת מופע וקרא לה

1. בפרויקט Classes אשר בסביבת הפיתוח, הוסף את שיטת המופע הציבורית הבאה **DistanceTo** אל המחלקה **Point**, בין הבנאים והמשתנים הפרטיים. השיטה מקבלת ארגומנט **Point** יחיד **other** ומחזירה משתנה מסוג **double**.

השיטה **DistanceTo** נראית כך:

```
class Point
{
   ...

   public double DistanceTo(Point other)
   {
   }
   ...
}
```

בשלבים הבאים עליך לערוך את גוף שיטת המקרה **DistanceTo** כדי שתחשב ותחזיר את המרחק בין האובייקט **Point** הנוכחי לבין האובייקט **Point** המועבר בתור פרמטר. כדי לעשות זאת, עליך לחשב את ההפרש בין קואורדינטות ה-**x** וקואורדינטות ה-**y**.

2. בשיטה **DistanceTo** הכרז על משתנה מקומי **xDif** מסוג **int**, ואתחל אותו להפרש בין **this.x** לבין **other.x**:

```
int xDiff = this.x – other.x;
```

3. הכרז על משתנה מקומי נוסף **yDiff** מסוג **int**, והתחל אותו להפרש שבין **this.y** לבין **other.y**.

השיטה **DistanceTo** נראית כך:

```
public double DistanceTo(Point other)
{
   int xDiff = x - other.x;
   int yDiff = y - other.y;
}
```

כדי לחשב את המרחק, תוכל להשתמש בשיטה המבוססת על משפט פיתגורס: חשב את השורש הריבועי של סכום הביטויים **xDiff** בריבוע ו-**yDiff** בריבוע. המחלקה **System.Math** כוללת את השיטה **Sqrt** שבעזרתה ניתן לחשב את השורש הריבועי.

4. הוסף את משפט ההחזרה (return) הבא לטובת ביצוע החישוב:

```
return Math.Sqrt(xDiff * xDiff + yDiff * yDiff);
```

על השיטה **DistanceTo** להיראות כעת כך:

```
public double DistanceTo(Point other)
{
   int xDiff = this.x - other.x;
   int yDiff = this.y - other.y;
   return Math.Sqrt(xDiff * xDiff + yDiff * yDiff);
}
```

כעת בדוק את השיטה DistanceTo.

5. חזור לשיטה **Entrance** שבמחלקה **Program**. לאחר המשפטים המכריזים על המשתנים **origin** ו-**bottomRight** מסוג **Point** ומאתחלים אותם, הכרז על משתנה **distance** מסוג **double**. אתחל את המשתנה לתוצאה שהתקבלה כאשר הפעלת את השיטה **DistanceTo** על האובייקט **origin** והעברת לה את האובייקט **bottomRight** בתור ארגומנט.

השיטה **Entrance** נראית כעת כך:

```
static void Entrance()
{
   Point origin = new Point();
   Point bottomRight = new Point(1024, 1280);
   double distance = origin.DistanceTo(bottomRight);
}
```

הערה

IntelliSense יציג את השיטה **DistanceTo** כאשר תקליד נקודה (.) לאחר **origin**.

6. הוסף משפט לשיטה **Entrance** שתפקידו לכתוב את ערך המשתנה **Distance** על המסך באמצעות השיטה **Consol.WriteLine**.

השיטה **Entrance** נראית כעת כך:

```
static void Entrance()
{
   Point origin = new Point();
   Point bottomRight = new Point(1024, 1280);
   double distance = origin.DistanceTo(bottomRight);
   Console.WriteLine("Distance is :{0}", distance);
}
```

7. בתפריט Debug, לחץ Start Without Debugging.

התוכנית נבנית ומופעלת.

8. ודא שהערך המופיע בחלון התצוגה הוא אכן 1640.605 (בקירוב). הקש Enter כדי לסגור את היישום ולחזור לסביבת הפיתוח.

הבנת שיטות ונתונים סטטיים

בתרגיל האחרון הפעלת את השיטה **Sqrt** מהמחלקה Class. אם חושבים על זה, הדרך שבה קראנו לשיטה הינו מוזר במקצת. למעשה, הפעלת את השיטה על המחלקה עצמה, ולא על אובייקט מסוג **Math**. אז מה קורה כאן וכיצד זה פועל?

לעיתים קרובות תגלה שלא כל השיטות משתייכות באופן טבעי למופע כלשהו של המחלקה. אלו הן שיטות עזר (utility methods) מבחינה זו שהן מספקות כלי עזר עצמאי מתוך המחלקה. השיטה **Sqrt** מהווה דוגמה לכך. אם **Sqrt** הייתה שיטת מופע (instance method) של **Math**, היית צריך ליצור אובייקט **Math** ועליו להפעיל את **Sqrt**:

```
Math m = new Math();
double d = m.Sqrt(42.24);
```

תהליך זה מסורבל מאוד. לאובייקט **Math** אין חלק בחישוב השורש הריבועי. כל נתוני הקלט שהשיטה **Sqrt** זקוקה להם נמצאים ברשימת הפרמטרים, ואילו התוצאה מוחזרת למשתמש באמצעות הערך המוחזר (return value) של השיטה. למעשה במקרה זה אין צורך במחלקות, ולכן הצבת ההגבלות הנובעות מיצירת מופע של **Math** והפעלת השיטה **Sqrt** אינה רעיון טוב כל-כך. **המחלקה** Math כוללת עזרים מתמטים רבים נוספים, כגון **Sin**, **Cos**, **Tan** ו-**Log**. המחלקה Math כוללת גם שדה עזר **PI** (פאי) אשר יכול היה לשמש אותנו בשיטה **Area** שבמחלקה **Circle**:

```
public double Area()
{
   return Math.PI * radius * radius;
}
```

בשפת #C יש להכריז על כל השיטות בתוך המחלקה. אולם, אם אתה מגדיר שיטה או שדה כסטטיים (**static**), תוכל לקרוא לשיטה, או לגשת לשדה בעזרת השם של המחלקה. אין צורך להשתמש במופע של המחלקה. להלן הדרך להכרזה על השיטה **Sqrt** מהמחלקה **Math**:

```
class Math
{
   public static double Sqrt(double d) { ... }
   ...
}
```

חשוב לזכור שלא מפעילים שיטה **סטטית** על אובייקט, או על מופע של מחלקה. כאשר אתה מגדיר שיטה **סטטית**, אין לה גישה לשדות המופע שהוגדרו עבור המחלקה. באפשרותה להשתמש בשדות סטטיים בלבד. בנוסף, היא יכולה להפעיל רק שיטות סטטיות אחרות במחלקה. עבור שיטות שאינן **סטטיות** (מופע) יש ליצור אובייקט אשר יקרא להן.

יצירת שדה משותף

כפי שנאמר בסעיף קודם, ניתן להשתמש במילת המפתח **static** גם להגדרת שדות. הדבר מאפשר לך ליצור **שדה משותף** (Shared Field) אחד לכל האובייקטים שנוצרו ממחלקה מסוימת. זכור ששדות שאינם סטטיים משתייכים באופן בלעדי לכל מופע של האובייקט. בדוגמה הבאה, ערך השדה הסטטי **NumCircles** מהמחלקה **Circle** עולה על ידי הבנאי **Circle**, בכל פעם שנוצר אובייקט **Circle** חדש.

```
class Circle
{
   public Circle() // default constructor
   {
      radius = 0.0;
      NumCircles++;
   }
   public Circle(double initialRadius) // overloaded constructor
   {
      radius = initialRadius;
      NumCircles++;
   }
   ...
   private double radius;
   public static int NumCircles = 0;
}
```

השדה **NumCircles** משותף לכל האובייקטים של **Circle**, ולכן המשפט **NumCircles++;** מגדיל את ערכו בכל פעם שנוצר מקרה חדש. כדי לגשת לשדה **NumCircles** עליך לציין את המחלקה **Circle**, ולא להשתמש במופע מסוים שלה. לדוגמה:

```
Console.WriteLine("Number of Circle objects: {0}",Circle.NumCircles);
```

טיפ

שיטות סטטיות מכונות גם **שיטות מחלקתיות**. אולם, **שדות סטטיים** לרוב אינם מכונים שדות מחלקתיים, אך לעיתים הם נקראים **משתנים סטטיים**.

יצירת שדה סטטי באמצעות מילת המפתח const

באפשרותך לקבוע ששדה כלשהו הינו סטטי, אולם ערכו אינו יכול להשתנות. עליך להוסיף את מילת המפתח **const** לפני שם השדה. **const** הוא קיצור של constant (קבוע). למרות שבהכרזה על שדה מסוג **const** אין צורך להשתמש במילת המפתח **static**, השדה הוא בכל זאת סטטי. אולם מסיבות שלא נדון בהן כאן, ניתן להכריז על שדה כ-**const** רק כאשר הוא מסוג **רשימה** (enum), סוג **פרימיטיבי** (primitive) או **מחרוזת** (string). להלן דוגמה כיצד המחלקה **Math** האמיתית מכריזה על **PI** כשדה מסוג **const** (זכור שיש יותר ספרות מימין לנקודה העשרונית):

```
class Math
{
   ...
   public const double PI = 3.14159265358979;
}
```

מחלקות סטטיות

מאפיין נוסף של שפת #C הוא היכולת להכריז על מחלקה כ**סטטית**, אשר יכולה לכלול חברים **סטטיים** בלבד. כלומר, כל האובייקטים שתיצור באמצעות מחלקה שכזו יחלקו בעותק יחיד של חברי המחלקה. מטרת המחלקה הסטטית להכיל שיטות ושדות עזר (utility) שונים. מחלקה **סטטית** אינה יכולה להכיל שיטות מופע או נתוני מופע, ואין הגיון בניסיון ליצור אובייקט ממחלקה שכזו באמצעות אופרטור **new**. למעשה, אינך יכול ליצור מקרה של אובייקט באמצעות מחלקה **סטטית**, ואם תנסה ־ המהדר יודיע על שגיאה. אם ברצונך לבצע אתחול כלשהו, מחלקה **סטטית** יכולה לכלול בנאי ברירת מחדל, כל עוד גם הוא מוגדר כ**סטטי**. כל סוג אחר של בנאי אינו חוקי ויגרום לשגיאה בשלב ההידור.

אם היית מגדיר גרסה משלך למחלקה **Math**, אשר כוללת חברים **סטטיים** בלבד, ייתכן שהיא הייתה נראית כך:

```
public static class Math
{
   public static double Sin(double x) {…}
   public static double Cos(double x) {…}
   public static double Sqrt(double x) {…}
   ...
}
```

חשוב לזכור שהמחלקה **Math** האמיתית אינה מוגדרת באופן כזה, כי היא כוללת מספר שיטות מופע (ואמנם, יכולנו ליצור אובייקט על ידי השימוש ב-new).

בתרגיל האחרון לפרק זה עליך להוסיף שדה מספרי **סטטי פרטי** למחלקה **Point**, ולאתחל את השדה ל-0. עליך להוסיף 1 לשדה זה בשני הבנאים. לבסוף, עליך לכתוב שיטה **סטטית פרטית** אשר תחזיר את הערך של השדה הסטטי הפרטי. שדה זה יאפשר לך לדעת כמה אובייקטים **Point** נוצרו עד כה.

כתוב חברי מחלקה סטטיים וקרא לשיטה סטטית

1. השתמש בסביבת הפיתוח כדי להציג את המחלקה **Point** בחלון Code and Text Editor.
2. בסופה של המחלקה Point, הוסף שדה **סטטי פרטי objectCount** מסוג **int**. אתחל אותו ל-0 בעת ההכרזה עליו.

   המחלקה **Point** נראית כעת כך:

```
class Point
{
   ...
   private static int objectCount = 0;
}
```

הערה

סדר מילות המפתח **private** ו-**static** אינו חשוב, אולם יש העדפה לכך ש-**private** תהיה ראשונה ו-**static** - שנייה.

3. הוסף משפט לשני הבנאים של **Point** אשר יגדיל את השדה **objectCount** באחד.

   בכל פעם שנוצר אובייקט, נקרא גם הבנאי שלו. כל עוד אתה מגדיל את **objectCount** בכל אחד מהבנאים, כולל בנאי ברירת המחדל, **objectCount** יכיל את מספר האובייקטים אשר נוצרו עד כה. אסטרטגיה זו ישימה רק מכיוון ש-**objectCount** הינו שדה סטטי משותף. אם **objectCount** היה שדה מופע (instance field), אזי לכל אובייקט היה שדה **objectCount** נפרד בעל ערך 1. המחלקה **Point** נראית כעת כך:

```
class Point
{
   public Point()
   {
      this.x = -1;
      this.y = -1;
      objectCount++;
   }
   public Point(int x, int y)
   {
      this.x = x;
      this.y = y;
      objectCount++;
   }
   ...
   private static int objectCount = 0;
}
```

שים לב שאינך יכול להוסיף את מילת המפתח **this** לפני שדות ושיטות **סטטיים**, כי אינם שייכים למופע הנוכחי של המחלקה, ולמעשה גם לא לכל מופע אחר.

השאלה שעולה היא, כיצד יכולים משתמשים במחלקה **Point** לדעת מהו מספר האובייקטים **Point** אשר נוצרו? השדה **objectCount** עודנו **פרטי** ואינו זמין מחוץ למחלקה. פתרון לא כל-כך טוב יהיה לאפשר גישה ציבורית לשדה **objectCount**. אסטרטגיה זו עלולה לפגוע בכימוס של המחלקה; כלומר, לא תוכל לדעת בוודאות שערך השדה נכון, כי כל גורם יוכל לשנותו. פתרון טוב יותר יהיה לספק שיטה **סטטית פרטית** אשר תחזיר את ערך השדה **objectCount**. את זה עליך לבצע כעת.

.4 הוסף שיטה סטטית פרטית בשם **ObjectCount** למחלקה **Point** אשר מחזירה ערך מסוג **int** אולם אינה מקבלת פרמטרים. בשיטה זו, החזר את ערך השדה **objectCount**.

המחלקה **Point** נראית כעת כך:

```
class Point
{
   public static int ObjectCount()
   {
      return objectCount;
   }
   ...
}
```

.5 הצג את המחלקה **Program** בחלון Code and Text Editor, ואתר את השיטה **Entrance**.

.6 הוסף משפט לשיטה **Entrance**, אשר מציג במסך את הערך שהשיטה **ObjectCount** החזירה מהמחלקה **Point**.

```
static void Entrance()
{
   Point origin = new Point();
   Point bottomRight = new Point(600, 800);
   double distance = origin.distanceTo(bottomRight);
   Console.WriteLine("Distance is :{0}", distance);
   Console.WriteLine("No of Point objects :{0}", Point.
     ObjectCount());
}
```

השיטה **ObjectCount** נקראת על ידי **Point**, כלומר שם המחלקה, ולא באמצעות שם המשתנה **Point**, כגון **Origin** או **bottomRight**. מכיוון ששני אובייקטים **Point** נוצרו עד רגע הקריאה ל-**ObjectCount**, השיטה אמורה להחזיר את הערך 2.

.7 בתפריט Debug לחץ Start Without Debugging.

.8 ודא שבחלון התצוגה אכן מופיע הערך 2, לאחר הודעה המציגה את ערך המשתנה **distance**. הקש Enter כדי לסיים את הפעלת התוכנית ולחזור ל- Visual Studio 2005.

מזל טוב! יצרת מחלקה (class) בהצלחה, והשתמשת בבנאים (constructor) כדי לאתחל את השדות שנמצאים בה. יצרת שיטות מופע (instance methods) ושיטות **סטטיות** (**static** methods), וקראת לסוגי השיטות הללו. כמו כן, יישמת שדות מופע ושדות **סטטיים**. למדת ליצור נגישות לשדות ולשיטות באמצעות מילת המפתח **public**, וכיצד להסתירם באמצעות מילת המפתח **private**.

☯ **אם ברצונך להמשיך לפרק הבא**

השאר את Visual Studio 2005 פעילה ועבור לפרק 8.

☯ **אם ברצונך לכבות כעת את Visual Studio 2005**

פתח את תפריט Menu ולחץ Exit. אם מופיעה תיבת דו-שיח Save, לחץ Yes.

פרק 7 – טבלה מסכמת

המשימה	צריך

להכריז על מחלקה.	לכתוב את מילת המפתח class, לאחריה את שם המחלקה, ולבסוף צמד סוגריים. השיטות והשדות מוגדרים בין שני סוגריים. לדוגמה:
	`class Point`
	`{`
	`  ...`
	`}`

להכריז על בנאי.	לכתוב שיטה אשר שמה זהה לשם המחלקה ושאינה מחזירה ערך מכל סוג (גם לא void). לדוגמה:
	`class Point`
	`{`
	`  public Point(int x, int y)`
	`  {`
	`  ...`
	`  }`
	`}`

לקרוא לבנאי.	להשתמש במילת המפתח new, ולהקצות לבנאי קבוצה מתאימה של פרמטרים. לדוגמה: `Point origin = new Point(0, 0);`

להכריז על שיטה **סטטית**.	לכתוב את מילת המפתח static לפני ההכרזה על השיטה. לדוגמה:
	`class Point`
	`{`
	`  public static int ObjectCount()`
	`  {`
	`  ...`
	`  }`
	`}`

לקרוא לשיטה **סטטית**.	לכתוב את שם המחלקה, לאחריה נקודה, ולבסוף את שם השיטה. לדוגמה: `int pointsCreatedSoFar = Point.ObjectCount();`

להכריז על שדה **סטטי**.	לכתוב את מילת המפתח static לפני ההכרזה על השדה. לדוגמה:
	`class Point`
	`{`
	`  ...`
	`  private static int objectCount;`
	`}`

פרק 8

הבנת ערכים והפניות

בסיום פרק זה, תוכל:

- להסביר את ההבדל בין סוג ערך (value) לסוג הפניה (reference).
- לשנות את אופן העברת הארגומנטים כפרמטרים של שיטה, באמצעות מילות המפתח **ref** ו-**out**.
- לארוז (box) ערך על ידי אתחול או הצבה במשתנה מסוג **object**.
- לפרוק (unbox) ערך באמצעות השלכת (casting) הייחוס לערך הארוז.

בפרק 7 למדת להכריז על מחלקות וליצור אובייקטים באמצעות מילת המפתח **new**. למדת גם לבנות אובייקט באמצעות בנאי. בפרק זה תלמד על ההבדלים המהותיים בין הסוגים הפרימיטיביים (primitive type) כגון **int**, לבין סוגי מחלקה (class type) כגון **Circle**.

העתקת משתני int ומחלקות

כל הסוגים הפרימיטיביים, כגון **int**, נקראים **סוגי ערך** (value type). בעת הכרזה על משתנה **int**, המהדר יוצר קוד אשר מקצה בלוק זיכרון גדול דיו להכלת מספר שלם (integer). משפט אשר מציב ערך (42 לדוגמה) במשתנה **int**, גורם להעתקת ערך זה אל הזיכרון המוקצה למשתנה.

הטיפול בסוגי מחלקה, כגון **Circle** (ראה פרק 7), הוא שונה. בעת הכרזה על משתנה **Circle**, המהדר **אינו** יוצר קוד המקצה בלוק זיכרון גדול דיו להכלת **Circle**, אלא מקצה פיסת זיכרון קטנה אשר יכולה להכיל את הכתובת של, או הפניה אל, בלוק זיכרון אחר המכיל את **Circle**. הזיכרון עבור האובייקט **Circle** מוקצה רק כאשר משתמשים במילת המפתח **new** כדי ליצור את האובייקט.

כעת מובן שסוגי ערך נקראים כך, כי הם מכילים ערכים בצורה ישירה, **וסוגי הפניה** (reference types) (כגון מחלקות) מכילים הפניות לבלוקים של זיכרון.

הערה

מתכנתים בעלי רקע תכנות של C או ++C עשויים לראות בסוגי הפניה אלה **מצביעים** (pointers). עם זאת, למרות שסוגי הפניה של #C דומים למצביעים מבחינות רבות, הם הרבה יותר שימושיים וגמישים. לדוגמה, ביישומי C או ++C ניתן לגרום למצביע להפנות כמעט לכל בלוק של זיכרון, תהיה תכולתו אשר תהיה. תכונה זו שימושית לעיתים, אולם לעיתים קרובות יותר היא הגורם לשגיאות תכנות שקשה לאתר. ב-#C כל ההפניות הן בעלות **סיווג נוקשה** (**strongly typed**). לא ניתן להכריז על משתנה הפניה אשר מפנה לסוג אחד (כגון **Circle**) ולאחר מכן להשתמש באותו משתנה כדי לגשת לבלוק זיכרון המכיל סוג אחר. יש הבדלים נוספים הנובעים מהדרך שבה סביבת ההרצה (CLR) מנהלת את הזיכרון. מאפיינים אלו מוסברים בפרק 13.

עקב הדרכים השונות שבאמצעותן סוגי הערך מאחסנים מידע, הם נקראים לעיתים **סוגים ישירים** (**direct types**), וסוגי הפניה נקראים לעיתים **סוגים עקיפים** (indirect types). חשוב להבין כראוי את ההבדל בין סוגי ערך לסוגי הפניה.

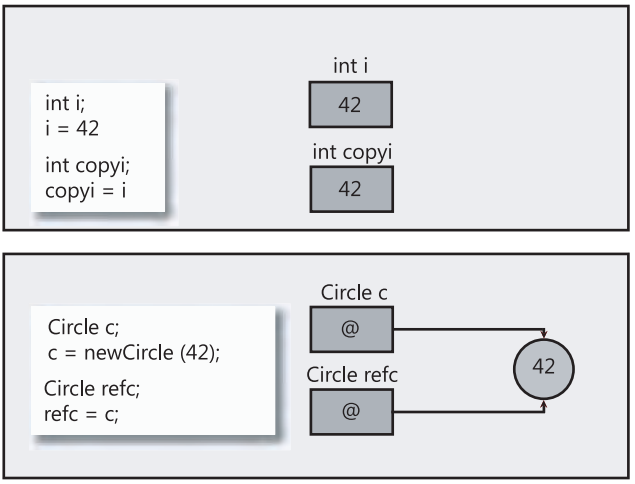
תאר לעצמך מצב שבו אתה מכריז על המשתנה **i** מסוג **int**, ומציב בו את הערך 42. אם תכריז על משתנה **int** נוסף בשם **copyi** ותציב בו את **i** , המשתנה **copyi** יכיל ערך זהה לזה של i (42), אולם העובדה שגם **i** וגם **copyi** מכילים את אותו הערך, אינה משנה את העובדה שיש שני עותקים של הערך 42: אחד בתוך **i** והשני בתוך **copyi**. אם תשנה את הערך של **i**, הערך של **copyi** לא ישתנה. הבה נעיין בקוד:

```
int i = 42;// declare and initialize i
int copyi = i;// copyi contains a copy of the data in i
i++;// incrementing i has no effect on copyi
```

אם תכריז על **c** בתור **Circle** (שם המחלקה) יהיו לפעולה השלכות שונות לגמרי. כאשר אתה מכריז על **c** כ-**Circle**, c יכול להפנות אל האובייקט **Circle**. אם תכריז על **Circle** נוסף בשם **refc**, גם הוא עשוי להפנות אל האובייקט **Circle**. אם תציב את **c** ב-**refc**, **refc** יפנה לאותו אובייקט שאליו מפנה **c**; יש אובייקט **Circle** אחד בלבד ושני המשתנים, **refc** ו-**c**, פונים אליו. הנה הקוד:

```
Circle c = new Circle(42);
Circle refc = c;
```

התרשים הבא ממחיש את שני המקרים:



ההבדלים שזה עתה הוצגו חשובים מאוד. המשמעות העיקרית של שוני זה בכך שהתנהגות הפרמטרים של שיטות תלויה בהיותם מסוג ערך (value) או מסוג הפניה (reference). בתרגיל הבא תלמד כיצד ההבדלים הללו באים לידי ביטוי.

השתמש בפרמטרים מסוג ערך ובפרמטרים מסוג הפניה

1. הפעל את Microsoft Visual Studio 2005.
2. פתח את הפרויקט **Parameters** הנמצא בתיקייה My Documents\Microsoft Press\Visual CSharp Step by Step\Chapter 8\Parameters.
3. הצג את קובץ המקור Pass.cs בחלון Code and Text Editor. אתר את המחלקה **Pass**. הוסף שיטה **סטטית ציבורית** בשם **value** למחלקה **Pass**. שיטה זו אמורה לקבל פרמטר **int** יחיד **param** מסוג ערך, ולהחזיר ערך **void**. גוף השיטה אמור להציב במשתנה **param** את הערך 42.

המחלקה **Pass** נראית כעת כך:

```
namespace Parameters
{
   class Pass
   {
      public static void Value(int param)
      {
         param = 42;
      }
   }
}
```

4. הצג את קובץ המקור Program.cs בחלון Code and Text Editor, ואתר את השיטה **Entrance** של המחלקה Class.

השיטה **Main** קוראת לשיטה **Entrance**, כאשר התוכנית מתחילה לפעול. כמוסבר בפרק 7, הקריאה לשיטה נמצאת בתוך בלוק **try** שלאחריו המטפל **catch**.

5. הוסף ארבעה משפטים לשיטה **Entrance** כדי לבצע את הפעולות הבאות:

א. הכרזה על משתנה **int** בשם **i** שמאותחל לערך 0.

ב. הצגת הערך של i במסך באמצעות השיטה **Console.WriteLine**.

ג. קריאה לשיטה **Pass.Value** עם **i** בתור ארגומנט.

ד. הצגת הערך של **i** במסך בשנית.

הקריאה ל-**Console.WriteLine** לפני ואחרי הקריאה ל-**Pass.Value** מאפשרת לדעת אם הקריאה ל-**Pass.Value** גרמה לשינוי הערך של **i**. השיטה **Entrance** נראית כעת כך:

```
static void Entrance()
{
   int i = 0;
   Console.WriteLine(i);
   Pass.Value(i);
   Console.WriteLine(i);
}
```

6. בתפריט Debug בחר Start Without Debugging כדי לבנות ולהפעיל את התוכנה.

7. ודא שהערך 0 מופיע פעמיים על המסך.

ההצבה בתוך **Pass.Value** בוצעה באמצעות העתקת הארגומנט, והארגומנט המקורי **i** אינו מושפע כלל.

8. הקש Enter כדי לסגור את היישום.

9. הצג את קובץ המקור WrappedInt.cs בחלון Code and Text Editor. הוסף למחלקה **WrappedInt** שדה מופע (instance) ציבורי מסוג **int** בשם **Number**.

המחלקה **WrappedInt** נראית כעת כך:

```
namespace Parameters
{
   class WrappedInt
   {
      public int Number;
   }
}
```

10. הצג את קובץ המקור Pass.cs בחלון Code and Text Editor. הוסף למחלקה **Pass** שיטה **סטטית ציבורית** בשם **reference**. המחלקה צריכה לקבל פרמטר אחד **param** מסוג **WrappedInt** ולהחזיר ערך מסוג **void**. גוף השיטה **Reference** מציב את הערך 42 ב-**param.Number**.

המחלקה **Pass** נראית כעת כך:

```
namespace Parameters
{
   class Pass
   {
      public static void Value(int param)
      {
         param = 42;
      }

      public static void Reference(WrappedInt param)
      {
         param.Number = 42;
      }
   }
}
```

11. הצג את קובץ המקור Program.cs בחלון Code and Text Editor. הוסף ארבעה משפטים נוספים לשיטה **Entrance**, אשר יבצעו את הפעולות הבאות:

א. הכרזה על משתנה מקומי **wi** מסוג **WrappedInt** ואתחולו לאובייקט חדש מסוג **WrappedInt** באמצעות קריאה לבנאי ברירת מחדל.

ב. הצגת הערך של **wi.Number** במסך.

ג. קריאה לשיטה **Pass.Reference** עם **wi** בתור ארגומנט.

ד. הצגת הערך של **wi.Number** במסך בשנית.

כמו קודם, הקריאות לשיטה **Console.WriteLine** מאפשרות לדעת האם הקריאה לשיטה **Pass.Reference** גרמה לשינוי הערך של **wi.Number**. השיטה **Entrance** נראית כעת כך:

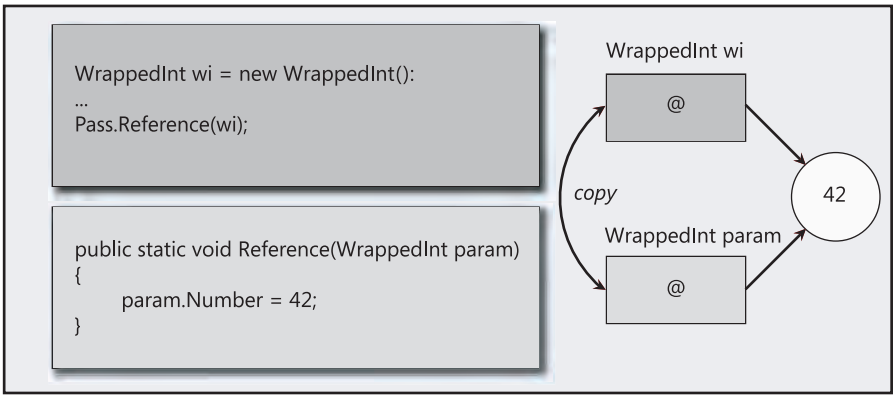
```
static void Entrance()
{
   int i = 0;
   Console.WriteLine(i);
   Pass.Value(i);
   Console.WriteLine(i);
   WrappedInt wi = new WrappedInt();
   Console.WriteLine(wi.Number);
   Pass.Reference(wi);
   Console.WriteLine(wi.Number);
}
```

12. בתפריט Debug בחר Start Without Debugging, כדי לבנות ולהפעיל את התוכנה.

גם לאחר פעולה זו הערך 0 מופיע במסך פעמיים, לפני ואחרי הקריאה ל-**Pass.Value**. עם זאת, שני הערכים הבאים, המייצגים את הערך של **wi.Number** לפני ואחרי הקריאה ל-**Pass.Reference**, הם 0 ו-42.

13. הקש Enter כדי לסגור את היישום.

בתרגיל האחרון, הערך של **wi.Number** מאותחל ל-0 על ידי הקוד שהמהדר יצר. המשתנה **wi** מכיל הפניה לאובייקט **WrappedInt** שזה עתה נוצר, ואשר מכיל ערך **int**. המשתנה **wi** מועבר כארגומנט לשיטה **Pass.Reference**. מכיוון ש-**WrappedInt** הינה מחלקה (מסוג הפניה), גם **wi** וגם **param** מתייחסים אל אותו אובייקט **WrappedInt**. כל שינוי בתוכן האובייקט באמצעות המשתנה **param** בשיטה **Pass.Reference** בא לידי ביטוי גם במשתנה **wi** שבסיום השיטה. התרשים הבא ממחיש מה קורה כאשר אובייקט **WrappedInt** מועבר בתור ארגומנט אל השיטה **Pass.Reference**:



הפרמטרים ref ו-out

בעת העברת ארגומנט לשיטה, הפרמטר המתאים מאותחל לעותק של הארגומנט. עובדה זו נכונה לגבי פרמטרים מסוג ערך (כגון **int**) ולגבי פרמטרים מסוג הפניה (כגון **WrappedInt**) כאחד. משמעות הדבר היא שאין שום אפשרות ששינוי בפרמטר ישפיע על הערך של הארגומנט שהועבר לו. לדוגמה, בקוד הבא הערך המוצג במסך הוא 42 ולא 43. השיטה **DoWork** מגדילה ב-1 עותק של הארגומנט (**arg**) ולא את הארגומנט עצמו:

```
static void DoWork(int param)
{
   param++;
}
static void Main()
{
   int arg = 42;
   DoWork(arg);
   Console.WriteLine(arg); // writes 42 not 43
}
```

בתרגיל האחרון ראינו שאם פרמטר של שיטה הוא מסוג הפניה, אז כל שינוי שנעשה באמצעות הפרמטר ישנה את הנתונים שהארגומנט שהועבר לשיטה מפנה אליהם. הנקודה המרכזית היא, שלמרות שהנתונים שההפניה מכוונת אליהם השתנו, הפרמטר עצמו לא השתנה והוא עדיין מפנה לאותו אובייקט. במילים אחרות, למרות שניתן לשנות את האובייקט שהארגומנט מפנה אליו על ידי האופרטור, לא ניתן לשנות את הארגומנט עצמו, כמו לדוגמה, כדי לגרום לכך שהוא יפנה לאובייקט אחר לגמרי. הביטחון הנובע מעובדה זו שימושי ביותר ותורם להפחתת מספר הבאגים בתוכנה. אולם לעיתים נרצה לכתוב שיטה שכן תשנה את הארגומנט. שפת C# מאפשרת לעשות זאת באמצעות מילות המפתח **ref** ו-**out**.

יצירת פרמטרים מסוג ref

הוספת מילת המפתח **ref** לפני פרמטר גורמת לכך שהפרמטר מהווה בעצמו שם בדוי של, או הפניה אל, הארגומנט עצמו ולא אל עותק של הארגומנט. משמעות השימוש במילת המפתח **ref** היא שכל פעולה המבוצעת על הפרמטר, מבוצעת גם על הארגומנט המקורי, כי שניהם מפנים אל אותו אובייקט. בעת הזנת ארגומנט בפרמטר **ref**, עליך להוסיף שוב את מילת המפתח **ref**, והפעם – לפני הארגומנט. תחביר זה נועד לתת ציון חזותי לעובדה שהארגומנט עשוי להשתנות. להלן הדוגמה הקודמת פעם נוספת,עם שימוש במילת המפתח **ref**:

```
static void DoWork(ref int param) // using ref
{
   param++;
}
static void Main()
{
   int arg = 42;
   DoWork(ref arg); // using ref
   Console.WriteLine(arg); // writes 43
}
```

מכיוון שלשיטה **DoWord** מועברת הפניה לארגומנט עצמו ולא אל עותק שלו, כל שינוי שנעשה באמצעות ההפניה גורם לשינוי גם בארגומנט המקורי, ולכן על המסך יופיע הערך 43.

החוק שאומר שעליך להציב ערך במשתנה לפני שתוכל להשתמש בו שימוש, תקף גם לגבי ארגומנטים מסוג **ref**. בדוגמה הבאה למשל, המשתנה **arg** אינו מאותחל, ולכן לא יהיה ניתן להדר את הקוד. הכישלון נובע מן העובדה שהביטוי **param++** שבתוך השיטה **DoWork** הוא למעשה **arg++**. ניתן לבצע את **arg++** רק אם arg מכיל ערך מוגדר:

```
static void DoWork(ref int param) // using ref
{
   param++;
}
static void Main()
{
```

```
    int arg; // not initialized
    DoWork(ref arg);
    Console.WriteLine(arg);
}
```

יצירת פרמטרים מסוג out

המהדר מוודא שהוצב ערך בפרמטר **ref** לפני הקריאה לשיטה. אולם ייתכנו מקרים בהם תרצה שהשיטה עצמה תבצע את האתחול של הפרמטר, לאחר העברת ארגומנט לא מאותחל אליה. פעולה זו מתאפשרת באמצעות מילת המפתח **out**.

מילת המפתח **out** דומה מאוד למילת המפתח **ref**. הוספת מילת המפתח **out** לפני הפרמטר תגרום לכך שהפרמטר יהיה למעשה שם בדוי עבור הארגומנט. בדומה לשימוש ב-**ref**, כל פעולה המבוצעת על הפרמטר תשפיע גם על הארגומנט המקורי. בעת העברת ארגומנט לפרמטר יש להוסיף לפני הארגומנט, בדומה ל-**ref**, את מילת המפתח out פעם נוספת.

מילת המפתח **out** היא קיצור של **output** (פלט). בעת העברת פרמטר **out** לשיטה, השיטה מציבה בו ערך. הדוגמה הבאה לא תעבור הידור, כי השיטה **DoWork** אינה מציבה ערך בפרמטר **param**:

```
static void DoWork(out int param)
{
   // Do nothing
}
```

הדוגמה הבאה, למשל, תעבור הידור כי הפרמטר **param** מקבל ערך:

```
static void DoWork(out int param)
{
   param = 42;
}
```

השיטה חייבת להציב ערך בפרמטר **out**, ולכן מותר לקרוא לה מבלי לאתחל את הארגומנט המועבר אליה. בסיום הקריאה לשיטה, הארגומנט צריך להיות בעל ערך. לדוגמה, הקוד הבא קורא לשיטה **DoWork** לטובת אתחול המשתנה **arg** אשר מוצג לאחר מכן במסך:

```
static void DoWork(out int param)
{
   param = 42;
}
static void Main()
{
   int arg; // not initialized
   DoWork(out arg);
   Console.WriteLine(arg); // writes 42
}
```

השתמש בפרמטר מסוג ref

1. חזור לפרויקט **Parameters** שבסביבת הפיתוח.
2. הצג את קובץ המקור Pass.cs בחלון Code and Text Editor.
3. ערוך את השיטה **Value** כדי שתקבל את הפרמטר **int** בתור פרמטר מסוג **ref**.

   השיטה **Value** נראית כעת כך:

```
class Pass
{
   public static void Value(ref int param)
   {
      param = 42;
   }
   ...
}
```

4. הצג את קובץ המקור Program.cs בחלון Code and Text Editor.
5. ערוך את המשפט השלישי בשיטה **Entrance**, כדי שהקריאה לשיטה **Pass.Value** תעביר את הארגומנט בתור פרמטר **ref**.

   השיטה **Entrance** נראית כעת כך:

```
class Application
{
   static void Entrance()
   {
      int i = 0;
      Console.WriteLine(i);
      Pass.Value(ref i);
      Console.WriteLine(i);
      ...
   }
}
```

6. בתפריט Debug בחר Start Without Debugging, כדי לבנות ולהפעיל את התוכנית.

   הפעם שני הערכים הראשונים שעל המסך יהיו 0 ו-42. תוצאה זו מוכיחה שהקריאה לשיטה **Pass.Value** שינתה את ערך הארגומנט **i**.
7. הקש Enter כדי לסגור את היישום.

הערה

ניתן להשתמש במילות המפתח **ref** ו-**out** גם עם פרמטרים מסוג הפניה (reference) ולא רק מסוג ערך (value). התוצאה זהה לחלוטין. הפרמטר הוא למעשה שם בדוי של הארגומנט. אם שינית את ההפניה של הפרמטר לאובייקט חדש, הארגומנט יפנה גם הוא אל אותו אובייקט חדש.

ארגון הזיכרון של המחשב

מחשבים משתמשים בזיכרון כדי לאחסן את התוכניות המופעלות, ואת הנתונים המשמשים תוכניות אלו. הבנת שיטת ארגון הנתונים בזיכרון יכולה לסייע בהבנת ההבדלים בין הסוגים ערך (value) לבין הפניה (reference).

מערכות הפעלה וסביבות הרצה (כגון CLR) מחלקות לעיתים קרובות את הזיכרון המשמש לאחסון נתונים לשני חלקים נפרדים, אשר כל אחד מהם מאורגן באופן שונה. שני חלקים אלה נקראים **מחסנית** (**stack**) ו**מצבור** (**heap**) והם משרתים מטרות שונות מאוד:

- בעת קריאה לשיטה, הזיכרון לאחסון הפרמטרים והמשתנים המקומיים של השיטה נלקח מהמחסנית. בסיום הרצת השיטה (כאשר היא מחזירה ערך או זורקת חריג), הזיכרון ששימש לאחסון הפרמטרים והמשתנים המקומיים מוחזר למחסנית בצורה אוטומטית, כדי שיהיה זמין בעת קריאה לשיטה אחרת.
- בעת יצירת אובייקט, שהוא מופע של מחלקה, באמצעות מילת המפתח **new** וקריאה לבנאי, הזיכרון המשמש לבניית האובייקט נלקח תמיד מהמצבור. קודם הראינו שניתן להפנות לאובייקט מסוים ממספר מקומות באמצעות משתני הפניה. כאשר כל ההפניות לאותו אובייקט מפסיקות מלהתקיים, הזיכרון אשר שימש לאחסון האובייקט מתפנה לשימוש חוזר, אך ייתכן שלא ישמש מייד לאחסון נתונים אחרים. פרק 13 עוסק בהרחבה בנושא של ניהול זיכרון.

הערה

כל סוגי ערך (value) נוצרים במחסנית. כל סוגי ההפניה (reference) שהינם אובייקטים, נוצרים במצבור (heap).

השמות **מחסנית** ו**מצבור** נגזרים משיטת ארגון הזיכרון על ידי סביבת ההרצה.

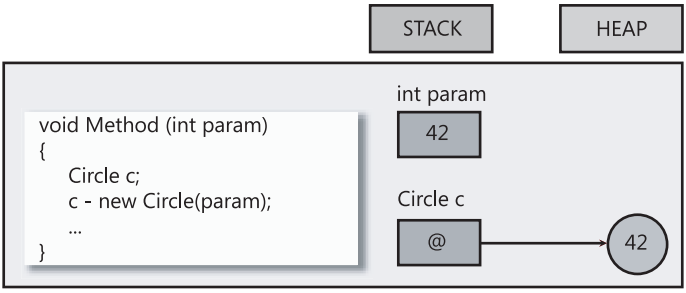
- זיכרון **מחסנית** (**stack**) מאורגן בדומה לארגזים המונחים זה על גבי זה. בעת קריאה לשיטה, כל פרמטר מאוחסן בארגז שנמצא בראש הערימה. כל אחד מהמשתנים המקומיים מקבל גם הוא ארגז, ואלה בתורם מונחים גם הם בראש הערימה. בסיום הפעלת השיטה, מפרקים את כל הארגזים מהערמה.
- זיכרון **מצבור** (**heap**) דומה למצבור גדול של ארגזים המפוזרים בחלל החדר, שאינם מונחים בצורה מסודרת זה על גבי זה. כל ארגז מסומן בתווית המציינת אם הוא בשימוש או לא. בעת יצירה של אובייקטים חדשים, סביבת ההרצה מחפשת ארגז פנוי (ריק), ומקצה אותו לאובייקט. סביבת ההרצה מתעדת את מספר ההפניות לכל ארגז (זכור ששני משתנים עשויים להפנות אל אותו אובייקט). עם היעלמותה של ההפניה האחרונה, סביבת ההרצה מסמנת את הארגז כ"לא בשימוש", ובשלב מסוים בעתיד תרוקן אותו לטובת שימוש חוזר.

שימוש במחסנית ובמצבור

מה קורה בעת הפעלת **השיטה** הבאה?

```
void Method(int param)
{
   Circle c;
   c = new Circle(param);
   ...
}
```

נניח שהערך שהועבר לפרמטר **param** הוא 42. חלק מהזיכרון, אשר גדול דיו לאחסון int, מוקצה מהמחסנית ומאותחל לערך 42. במסגרת השיטה, חלק נוסף מהזיכרון, גדול דיו להכיל הפניה, מוקצה מהמחסנית גם כן, אולם אינו מאותחל עבור המשתנה **c** מסוג **Circle**. לאחר מכן, חלק מהזיכרון, גדול מספיק לאחסון האובייקט **Circle**, מוקצה מתוך המצבור. זו למעשה הפעולה שמבצעת מילת המפתח **new**. הבנאי **Circle** מופעל כדי להמיר את זיכרון המצבור הגולמי לאובייקט מסוג **Circle**, ובמשתנה c מוצבת הפניה לאובייקט **Circle** החדש. התרשים הבא ממחיש את המצב:



בשלב זה חשוב לשים לב לשני דברים:

1. למרות שהאובייקט עצמו מאוחסן במצבור, ההפניה אליו (המשתנה c) מאוחסנת במחסנית.
2. זיכרון מצבור אינו אינסופי. כאשר זיכרון המצבור הפנוי "אוזל", האופרטור new יזרוק את החריג **OutOfMemoryException** והאובייקט לא ייווצר.

הערה

הבנאי **Circle** יכול גם הוא לזרוק חריג. כאשר הוא עושה זאת, הזיכרון שהוקצה לאובייקט **Circle** ישוחרר, והערך שמוחזר על ידי הבנאי תהיה הפניה ריקה (null).

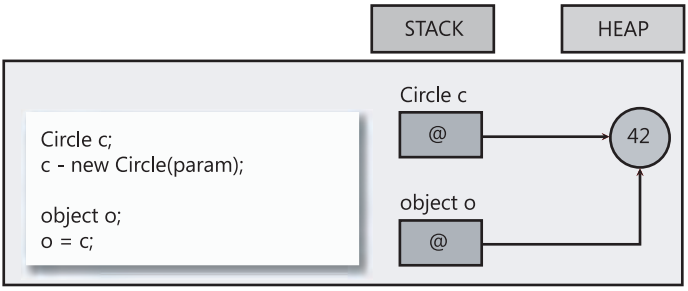
בסיום פעולה, תחום ההכרזה (scope) של הפרמטרים והמשתנים המקומיים נגמר. הזיכרון שהוקצה למשתנה **c** והזיכרון שהוקצה למשתנה **param** משוחררים ומוחזרים למחסנית. סביבת ההרצה מבחינה בכך שאין יותר הפניות לאובייקט **Circle**, ובשלב מסוים תפעל להשבת הזיכרון למצבור (ראה פרק 13).

המחלקה System.Object

המחלקה **Object** שבמרחב השמות **System** הינה אחת מסוגי ההפניה החשובים ביותר במערכת NET. Framework. כדי להבין את מלוא החשיבות של המחלקה **System.Object** עליך לדעת מהי **הורשה** (inheritance) המוסברת בהרחבה בפרק 12. בשלב זה, קבל כנתון שכל המחלקות הן סוג מיוחד של **System.Object**, ושניתן להשתמש באובייקט זה כדי ליצור משתנה שיכול להתאים עצמו לכל סוג הפניה שהוא. חשיבות המחלקה **System.Object** גדולה עד כדי כך שבשפת #C ניתן להשתמש במילת המפתח **Object** כתחליף ל-**System.Object**. בקוד אתה יכול להשתמש ב-**Object** או לכתוב **System.Object**. המשמעות זהה לחלוטין.

בדוגמה הבאה, המשתנים **c** ו-**o** מתייחסים לאותו אובייקט **Circle**. העובדה שהסוג של **c** הוא **Circle** בעוד שהסוג של **o** הוא **object** (כלומר **System.Object**) יוצרת למעשה שתי נקודות מבט שונות על אותו אובייקט.

```
Circle c;
c = new Circle(42);

object o;
o = c;
```

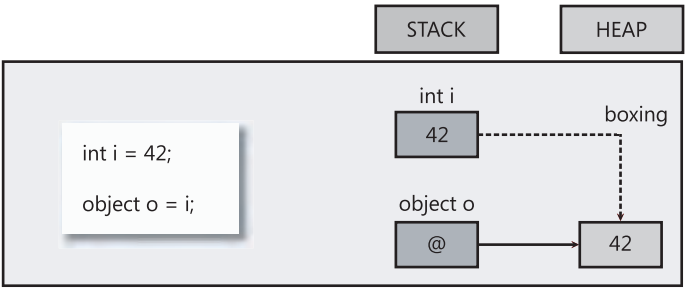


אריזה

כפי שראית, משתנים מסוג **object** יכולים להפנות לכל אובייקט מכל סוג הפניה. כלומר, הם מסוגלים להפנות לכל מופע של מחלקה. משתנים מסוג **object** יכולים להפנות גם לסוג **ערך** (value). לדוגמה, שני המשפטים הבאים מאתחלים את המשתנה **i** (מסוג **int**, סוג ערך) ל-42, ולאחר מכן מאתחלים את המשתנה **o** (מסוג **object**, סוג הפניה) לערך של **i**:

```
int i = 42;
object o = i;
```

המשפט השני גורם לתוצאות עדינות יותר. זכור ש-**i** הוא סוג ערך שמתקיים במחסנית. אם ההפניה שב- **o** מפנה ישירות ל-**i**, היא מפנה למעשה אל מחסנית. כל ההפניות מפנות לאובייקט במצבור ולכן, פיסת זיכרון מהמצבור מוקצית לטובת אחסון עותק מדויק של הערך המאוחסן במשתנה **i**, וההפניה שב-**o** מפנה אל אותו עותק. יצירה של הפניות אל פריטים במחסנית פוגעת באופן משמעותי ביציבות של סביבת ההרצה, ויוצרת פגם אפשרי באבטחה, ולכן הפעולה אינה מותרת. העתקה האוטומטית של הפריט במחסנית אל מצבור נקראת **אריזה** (**boxing**). התרשים הבא מדגים את תוצאות התהליך:



חשוב

שינוי הערך המקורי של משתנה לא יגרום לשינוי הערך שהועתק למצבור, כי הוא רק העתק.

פריקה

מכיוון שמשתנה מסוג **object** יכול להפנות אל עותק ארוז של ערך כלשהו, אין זה בלתי סביר שתוכל לגשת לערך ארוז זה דרך המשתנה. תוכל להניח שניתן לגשת לערך הארוז int אשר המשתנה **o** מפנה אליו באמצעות משפט הצבה פשוט כגון זה:

```
int i = o;
```

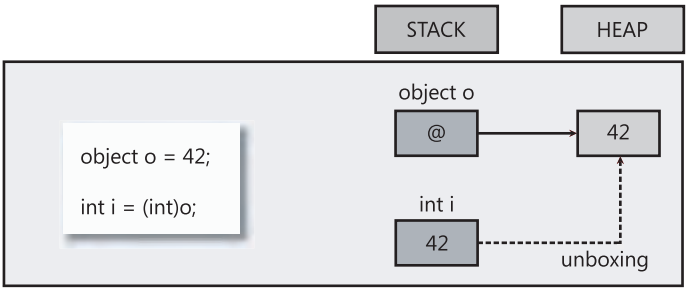
הדבר אסור, וכתיבת משפט זה תגרור שגיאת הידור. אם תחשוב על זה, יש הגיון בעובדה שלא ניתן להשתמש במשפט int i = o; אחרי הכל, **o** יכול למעשה להפנות לכל פריט שהוא ולא רק ל-**int**. נסה לחשוב מה היה קורה בקוד הבא, אם היה ניתן להשתמש במשפט:

```
Circle c = new Circle();
int i = 42;
object o;
o = c; // o refers to a circle
i = o; // what is stored in i?
```

כדי לקבל את ערך העותק הארוז, עליך להשתמש בתהליך בשם **השלכה** (**cast**) אשר נועד לבדוק אם ניתן להמיר בצורה בטוחה סוג אחד למשנהו, ולאחר מכן מבצע את ההמרה. לפני המשתנה מסוג **object** עליך להוסיף את שם הסוג בסוגריים, כפי שמוצג להלן:

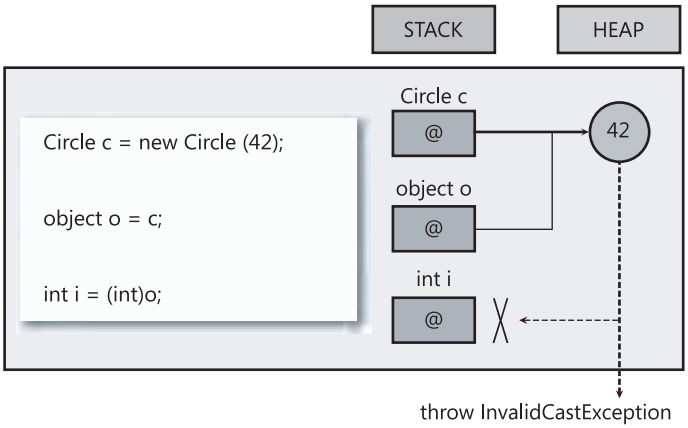
```
int i = 42;
object o = i; // boxes
i = (int)o; // compiles okay
```

תוצאות ההשלכה עדינות. המהדר מבחין בכך שציינת את הסוג **int** עבור האירוע. לאחר מכן, המהדר יוצר קוד שנועד לבדוק לאן בדיוק **o** מפנה את התוכנית בסביבת ההרצה. למעשה הוא יכול להפנות לכל דבר. העובדה שבהשלכה מצוין שהוא מפנה ל-**int**, אינה אומרת שזה מה שקורה בפועל. אם **o** באמת מפנה ל-**int** הארוז והכל מתאים, ההשלכה תפעל בהצלחה והקוד שהמהדר ייצור ישלוף את הערך מה-**int** הארוז. שים לב שבדוגמה זו הערך הארוז משמש כדי לאתחל את **i**. תהליך זה מכונה **פריקה** (**unboxing**), והתרשים הבא ממחיש כיצד הוא מתבצע:



אם **o** אינו מפנה ל-**int** ארוז, יש חוסר התאמה בין הסוגים וההשלכה נכשלת. הקוד שהמהדר יצר זורק חריג **InvalidCastException** בזמן ההפעלה. הנה דוגמה להשלכת פריקה שנכשלה:

```
Circle c = new Circle(42);
object o = c; // doesn't box because Circle is a class
int i = (int)o; // compiles okay, but throws at runtime
```



הערה

הסוג שמציינים בהשלכת הפריקה חייב להתאים בדיוק לסוג הארוז. לדוגמה, אם הסוג הארוז הוא העתק של **int** ותנסה לפרוק אותו לערך **long**, תקבל חריג **InvalidCastException**. העובדה שקיימת המרה מובנית פשוטה מ-**int** ל-**long** אינה רלוונטית. חייבת להיות התאמה מדויקת.

נתרגל את השימוש באריזה ובפריקה בשלב מאוחר יותר. זכור שאריזה ופריקה הם תהליכים בזבזניים, כי דרושות בדיקות מרובות והקצאה של זיכרון מחסנית נוסף. יש שימושים לאריזה, אולם שימוש לא זהיר בה עשוי לסכן את התוכנית ואף לפגוע בביצועים שלה. בפרק 17 תלמד על תחליפים לאריזה.

מצביעים וקוד לא בטוח

סעיף זה מיועד להרחבת הידע בלבד, והוא מכוון בעיקר למתכנתים מרקע תכנות של C ו-++C. קוראים שזו שפת התכנות הראשונה שלהם יכולים לדלג ולעבור לסעיף הבא.

אם עסקת בעבר בשפות התכנות C ו-++C, הרבה מהכתוב על הפניות לאובייקטים עשוי להיות מוכר לך. למרות שגם לשתי שפות תכנות אלו אין סוגי הפניה המוגדרים בבירור, שתיהן מכילות מבנה המבצע פעולה דומה – **מצביעים** (pointers).

מצביע הוא משתנה אשר מכיל את הכתובת של, או הפניה ל-, פריט בזיכרון שנמצא במחסנית או במצבור. תחביר מיוחד משמש לזיהוי המשתנה בתור מצביע. לדוגמה, המשפט הבא מכריז על המשתנה **pi** כמצביע על שלם (integer):

```
int *pi;
```

למרות שהמשתנה **pi** מוכרז בתור מצביע, למעשה אין הוא מצביע לשום מקום עד שמאתחלים אותו. לדוגמה, כדי להשתמש ב-**pi** כדי להצביע על המשתנה השלם **i**, ניתן להשתמש במשפטים הבאים, בנוסף לאופרטור **&**, אשר מחזיר את כתובת המשתנה:

```
int *pi;
int i = 99;
...
pi = &i;
```

ניתן לגשת לערך ולשנות את הערך המאוחסן במשתנה באמצעות המצביע **pi**. הנה כך:

```
*pi = 100;
```

קוד זה מעדכן את ערך המשתנה **i** ל-100, מכיוון ש-**pi** מצביע לאותו שטח בזיכרון שאליו מצביע המשתנה **i**.

אחת הבעיות העיקריות אשר עומדות בפני מפתחים הלומדים C ו-++C היא הבנת התחביר המשמש את המצביעים. לאופרטור * יש לפחות שתי משמעויות, בנוסף להיותו אופרטור ההכפלה, ולעיתים קרובות יש בלבול רב בין השימוש ב-**&** לשימוש ב-*. בעיה נוספת הנובעת מנושא המצביעים היא שקל להצביע על כתובת בלתי-חוקית, או לא להצביע בכלל, ולאחר מכן לנסות לפנות לנתונים שהמשתנה מצביע אליהם. התוצאה תהיה חסרת ערך או תוכנית שתיכשל עקב שגיאה, כי מערכת ההפעלה תזהה ניסיון גישה לכתובת בלתי-חוקית בזיכרון. יש פגמים נוספים שגורמים לכך שהקוד לא יהיה בטוח, ואמנם, תקלות רבות בתוכניות נגרמות מניהול לא תקין של מצביעים; סביבות מסוימות (Microsoft Windows אינה ביניהן) אינן מבצעות בדיקה כדי לוודא שהמצביע אינו מפנה לשטח בזיכרון אשר משמש תהליך אחר, וכך הן מסכנות תוכניות ובעיקר נתונים שעשויים להיות אישיים או סודיים.

בשפת #C הוסיפו את בדיקת ההפניה למשתנים כדי להימנע מן הבעיות הללו. ניתן כמובן להמשיך ולהשתמש במצביעים גם ב-#C, אולם עליך לסמן את הקוד במילת המפתח **unsafe** (**לא בטוח**). מילת מפתח זו משמשת לסימון בלוק של קוד או שיטה שלמה, כדי לציין שהקוד אינו בטוח:

```
public static void Main(string [] args)
{
   int x = 99, y = 100;
   unsafe
   {
      swap (&x, &y);
   }
   Console.WriteLine("x is now {0}, y is now {1}", x, y);
}

public static unsafe void swap(int *a, int *b)
{
   int temp;
   temp = *a;
   *a = *b;
   *b = temp;
}
```

בעת הידור תוכנית המכילה קוד שאינו בטוח, עליך לציין זאת באמצעות האפשרות **/unsafe**.

לקוד שאינו בטוח יש גם השלכות על אופן ניהול הזיכרון. אובייקטים שנוצרו במסגרת קוד unsafe (לא בטוח) הם **לא מנוהלים** (unmanaged).

בפרק זה למדת על הבדלים חשובים בין סוגי ערך (value types) המכילים את הערך באופן ישיר במחסנית (stack) ובין סוגי הפניה (reference types) המפנים באופן עקיף למצבור (heap). למדת גם להפעיל את מילות המפתח **ref** ו-**out** על פרמטרים של שיטה כדי לגשת לארגומנט. ראית כיצד הצבת ערך (42 למשל) במשתנה מהמחלקה **System.Object** תגרום לכך שהמשתנה יפנה לעותק ארוז של הערך במצבור. כמו כן, ראית כיצד הצבת משתנה מסוג ערך במשתנה מהמחלקה **System.Object** תגרום לכך שהעותק הארוז יפנה למשתנה מסוג ערך (**int** למשל).

☯ **אם ברצונך להמשיך לפרק הבא**

השאר את Visual Studio 2005 פעילה ועבור לפרק 9.

☯ **אם ברצונך לכבות כעת את Visual Studio 2005**

פתח את תפריט Menu ולחץ Exit. אם מופיעה תיבת דו-שיח Save, לחץ Yes.

פרק 8 – טבלה מסכמת

המשימה	צריך

להעתיק משתנה מסוג ערך (value type variable).	לבצע העתקה. המשתנה הוא מסוג ערך, ולכן לאחר הפעולה יהיו ברשותך שני עותקים של אותו הערך. לדוגמה:
	`int i = 42;`
	`int copyi = i;`

להעתיק משתנה מסוג הפניה (reference type variable).	לבצע העתקה. המשתנה הוא מסוג הפניה, ולכן לאחר הפעולה יהיו ברשותך שתי הפניות אל אותו אובייקט. לדוגמה:
	`Circle c = new Circle(42);`
	`Circle refc = c;`

להעביר ארגומנט לפרמטר מסוג **ref**.	להוסיף את מילת המפתח **ref** לפני הארגומנט. הפעולה תגרום לכך שהפרמטר יהיה שם המייצג את הארגומנט, ולא העתק שלו. לדוגמה:
	`static void Main()`
	`{`
	`int arg = 42;`
	`DoWork(ref arg);`
	`Console.WriteLine(arg);`
	`}`

להעביר ארגומנט לפרמטר מסוג **out**.	להוסיף את מילת המפתח **out** לפני הארגומנט. הפעולה תגרום לכך שהפרמטר יהיה שם המייצג את הארגומנט, ולא העתק שלו. לדוגמה:
	`static void Main()`
	`{`
	`int arg = 42;`
	`DoWork(out arg);`
	`Console.WriteLine(arg);`
	`}`

לארוז (box) ערך.	להציב ערך במשתנה מסוג object. לדוגמה: `object o = 42;`
לפרוק (unbox) ערך.	השלך (cast) את ההפניה לאובייקט המתייחסת לערך הארוז על הסוג של הערך. לדוגמה: `int i = (int)o;`

פרק 9

יצירת סוגי ערך באמצעות רשימות ומבנים

בסיום פרק זה, תוכל:

- להכריז על סוג רשימה (enumeration).
- ליצור סוג רשימה ולהשתמש בו.
- להכריז על סוג מבנה (structure).
- ליצור סוג מבנה ולהשתמש בו.

בפרק 8 למדת על שני סוגים חשובים של משתנים בשפת #C: סוגי ערך (value types) וסוגי הפניה (reference types). משתנה מסוג ערך מכיל את ערכו ישירות במחסנית, בעוד שמשתנה מסוג הפניה מכיל הפניה אל אובייקט שנמצא במצבור. בפרק 7 למדת לכתוב מחלקות משלך, ובכך ליצור סוגי הפניה משלך. בפרק זה תלמד כיצד לכתוב סוגי ערך משלך.

שפת #C תומכת בשני סוגי ערך שונים: רשימות (enumerations) ומבנים (structures).

רשימות ושימושיהן

נניח שברצונך לייצג בתוכנית את עונות השנה. תוכל להשתמש במספרים השלמים 0, 1, 2 ו-3 כדי לייצג את האביב, הקיץ, הסתיו והחורף, בהתאמה. שיטה זו אמנם טובה, אולם אינה אינטואיטיבית במיוחד. אם תשתמש בערך השלם 0 בקוד, לא יהיה זה מובן מאליו שהוא מייצג את האביב, למשל. כמו כן, הקוד לא יהיה יציב במיוחד כי ניתן להשתמש בכל סדרה אחרת של ארבעה מספרים שלמים, ולאו דווקא ב-0, 1, 2 ו-3. שפת #C נותנת לנו פתרון טוב יותר. באפשרותך להשתמש במילת המפתח enum כדי ליצור סוג רשימה שערכיה מוגבלים לקבוצת שמות סמליים.

הכרזה על סוג רשימה

להלן אופן ההכרזה על סוג רשימה (enumeration) בשם **Season**, אשר ערכיה המילוליים מוגבלים לשמות הסמליים **Spring**, **Summer**, **Fall** ו-**Winter**:

```
enum Season { Spring, Summer, Fall, Winter }
```

השמות הסמליים צריכים להופיע בין צמד סוגריים מסולסלים ברשימה המופרדת על ידי פסיקים. באופן פנימי, הרשימה מייחסת מספר שלם לכל פריט שנמצא בה. על פי ברירת המחדל, המספור מתחיל מ-0 עבור הפריט הראשון ומתקדם ב-1 בכל פריט עוקב.

שימוש ברשימה

לאחר הכרזה על סוג הרשימה, תוכל להשתמש בשם שהגדרת באופן זהה לשימוש בכל שם אחר. אם שם הרשימה הוא **Season**, תוכל ליצור משתנים מסוג **Season**, שדות מסוג **Season** ופרמטרים מסוג **Season**, כמוצג בדוגמה הבאה:

```
enum Season { Spring, Summer, Fall, Winter }
class Example
{
   public void Method(Season parameter)
   {
      Season localVariable;
      ...
   }
   private Season currentSeason;
}
```

סוג רשימה הוא סוג ערך. משתני רשימה מאוחסנים במחסנית (stack) (ראה פרק 8). אין זה מפתיע, בהתחשב בעובדה שסוג ערך מהווה בסיס לסוג רשימה.

כדי להשתמש בערכים של משתני רשימה, צריך להציב בהם ערך לפני השימוש. במשתני רשימה ניתן להציב ערכים חוקיים בלבד. לדוגמה:

```
Season colorful = Season.Fall;
Console.WriteLine(colorful); // writes out 'Fall'
```

שים לב שעליך לכתוב **Season.Fall** ולא רק **Fall**. כל השמות הליטרלים של הרשימה פועלים בתחום ההכרזה של סוג הרשימה בלבד. כלל זה שימושי ביותר כי הוא מאפשר לסוגי רשימות שונים לכלול ליטרלים בעלי אותו שם. שים לב, שבעת הצגת הערכים של הרשימה באמצעות השיטה **Console.WriteLine**, המהדר יוצר קוד אשר מדפיס את שם הליטרל שערכו תואם לערך המשתנה המתאים.

משמעות הדבר היא, שניתן להמיר משתנה רשימה למחרוזת המייצגת את ערכו הנוכחי, באמצעות השיטה **ToString** המובנית בכל סוגי הרשימות, באופן תורשתי מהסוג **System.Enum**. לדוגמה:

```
string name = colorful.ToString();
Console.WriteLine(name); // also writes out 'Fall'
```

ניתן גם לקבל את הערך השלם שביסודו של משתנה הרשימה. כדי לעשות זאת, עליך להשליך (cast) את משתנה הרשימה על הסוג שביסודו. זכור את ההסבר על הפריקה בפרק 8, שעל פיו השלכה של סוג גורמת להמרה של הנתונים מסוג אחד למשנהו, כל עוד ההמרה חוקית ומשמעותית. לדוגמה, קטע הקוד הבא יגרום להדפסת הערך 2 ולא **Fall** (**אביב** הוא 0, **קיץ** 1, **סתיו** 2 **וחורף** 3):

```
enum Season { Spring, Summer, Fall, Winter }
...
Season colorful = Season.Fall;
Console.WriteLine((int)colorful); // writes out '2'
```

חלק גדול מהאופרטורים הסטנדרטיים שניתן להפעיל על משתנים שלמים ניתן להפעיל גם על משתני רשימה, מלבד האופרטורים **bitwise** ו-**shift** המוסברים בפרק 15. לדוגמה, ניתן להשוות בין שני משתני רשימה מאותו סוג באמצעות האופרטור ==, כדי לבדוק אם הם זהים או לא, וגם ניתן לבצע פעולות אריתמטיות על משתני רשימה, למרות שהתוצאה לא תמיד תהיה בעלת משמעות.

בחירת הערכים הליטרלים של הרשימה

לכל רשימה יש קבוצה של ערכים שלמים הקשורים לפריטים הכלולים בה, הראשון ביניהם הוא 0 המציין את הפריט הראשון. באפשרותך לקשר בין קבוע שלם (כגון 1) לבין ליטרל מהרשימה (כגון **Spring**), כמוצג בדוגמה הבאה:

```
enum Season { Spring = 1, Summer, Fall, Winter }
```

חשוב

הערך השלם שאתה קושר אליו את הליטרל מהרשימה חייב להיות ערך שלא ישתנה בזמן ההידור (כגון 1). כלומר, הוא חייב להיות קבוע אשר ערכו אינו תלוי בפעולת ההידור (בניגוד לקריאה לשיטה, למשל).

אם אינך מקצה לליטרל ערך שלם קבוע, המהדר יקצה לו ערך הגדול באחד מערכו של הליטרל שלפניו, חוץ מהליטרל הראשון שמקבל מהמהדר את ערך ברירת המחדל 0. בדוגמה האחרונה, הערכים המייצגים את הליטרלים **Spring**, **Summer**, **Fall** ו-**Winter** הם 1, 2, 3 ו-4.

באפשרותך לקבוע את אותו ערך מספרי ליותר מאשר ליטרל אחד ברשימה. לדוגמה, בבריטניה אומרים **Autumn** ולא **Fall**. ניתן להתאים את הרשימה לכל אחד מהמקרים:

```
enum Season { Spring, Summer, Fall, Autumn = Fall, Winter }
```

לפיכך, הבדיקה הבאה תהיה חיובית :

```
Season seasonUSA = Season.Fall;
Season seasonUK = Season.Autumn;
if (seasonUSA == seasonUK)
Console.Write("The same Season");
```

בחירת הסוג שבבסיס הרשימה

בעת ההכרזה על סוג רשימה, הליטרלים שכלולים בה מקבלים ערכים מסוג **int**. במילים אחרות, ברירת המחדל של הסוג שבבסיס הרשימה היא int. ניתן לבסס את הרשימה גם על סוגים אחרים של מספרים שלמים. לדוגמה, כדי להכריז שסוג הערך עבור הרשימה **Season** הוא **short** במקום **int**, ניתן לכתוב את המשפט הבא:

```
enum Season : short { Spring, Summer, Fall, Winter }
```

הסיבה העיקרית לעשות זאת היא חיסכון בזיכרון. ערך **int** תופס יותר מקום בזיכרון מאשר ערך **short**, ומכיוון שאינך זקוק לכל טווח הערכים של **int**, עדיף להשתמש בסוג נתונים קטן יותר.

ניתן לבסס רשימה על כל אחד משמונת סוגי המספרים השלמים: **byte**, **sbyte**, **short**, **ushort**, **int**, **uint**, **long** או **ulong**. מספר הליטרלים שברשימה חייב להיות קטן או שווה לטווח המספרים של סוג הבסיס הנבחר. לדוגמה, אם בחרת לבסס את הרשימה על סוג הנתון **byte**, אינך יכול לכלול ברשימה יותר מ-256 ליטרלים (החל מ-0).

כעת, לאחר שלמדת כיצד ליצור סוג רשימה, הגיע הזמן להשתמש בו.

בתרגיל הבא, תעבוד עם יישום מסוף (consol) כדי להכריז על מחלקת רשימה המייצגת את חודשי השנה, ולהשתמש בה.

יצירת סוג רשימה ושימוש בו

1. הפעל את סביבת הפיתוח Microsoft Visual Studio 2005.
2. פתח את הפרויקט **StructsAndEnums** שבתיקייה My Documents\Microsoft Press\Visual CSharp Step by Step\Chapter 9\StructsAndEnums.
3. הצג את קובץ המקור Month.cs בחלון Code and Text Editor.

   הקובץ מכיל מרחב שמות ריק בשם **StructsAndEnums**.
4. הוסף סוג רשימה בשם **Month** במרחב השמות **StructsAndEnums**, אשר מפרט את חודשי השנה.

   12 הליטרלים שברשימה **Month** הם ינואר עד דצמבר. הרשימה **Month** נראית כך:

```
namespace StructsAndEnums
{
   enum Month
   {
      January, February, March, April,
      May, June, July, August,
      September, October, November, December
   }
}
```

5. הצג את קובץ המקור Program.cs בחלון Code and Text Editor.

בדומה לתרגילים בפרקים הקודמים, השיטה **Main** קוראת לשיטה Entrance ולוכדת חריגים (אם יש חריגים שצריך ללכוד).

6. בחלון Code and Text Editor, הוסף משפט לשיטה **Entrance** המכריז על משתנה מסוג **Month** בשם **first** ומאתחלת אותו ל-**Month.January**. הוסף משפט המציג את הערך של המשתנה first על המסך.

השיטה **Entrance** נראית כך:

```
static void Entrance()
{
   Month first = Month.January;
   Console.WriteLine(first);
}
```

הערה

נאשר תקליד את הנקודה שאחרי **Month**, רכיב Intellisense יציג באופן אוטומטי את כל הערכים שברשימה **Month**.

7. בתפריט Debug בחר Start Without Debugging.

סביבת הפיתוח תבנה ותפעיל את התוכנית. ודא שהמילה "January" מוצגת במסך.

8. הקש Enter כדי לסגור את התוכנית ולחזור לסביבת הפיתוח.

9. הוסף שני משפטים לשיטה Entrance, אשר מגדילים את ערך המשתנה first באחד ומציגים את ערכו החדש במסך.

השיטה **Entrance** נראית כך:

```
static void Entrance()
{
   Month first = Month.January;
   Console.WriteLine(first);
   first++;
   Console.WriteLine(first);
}
```

10. בתפריט Debug בחר Start Without Debugging.

סביבת הפיתוח תבנה ותפעיל את התוכנית. ודא שהמילים "January" ו-"February" מוצגות במסך. שים לב כיצד הפעלת אופרטור מתמטי (כגון אופרטור ההגדלה) על משתנה רשימה יכולה לשנות את הערך השלם הפנימי של המשתנה. כאשר הוא מוצג, מופיע הערך הליטרלי המקביל לערך השלם.

11. הקש Enter כדי לסגור את התוכנית ולחזור לסביבת הפיתוח.

12. שנה את המשפט הראשון בשיטה **Entrance**, כדי שיאתחל את המשתנה **first** ל-**Month.December**.

השיטה **Entrance** נראית כך:

```
static void Entrance()
{
   Month first = Month.December;
   Console.WriteLine(first);
   first++;
   Console.WriteLine(first);
}
```

13. בתפריט Debug בחר Start Without Debugging.

סביבת הפיתוח תבנה ותפעיל את התוכנית. הפעם המילה "December" תוצג במסך, ולאחריה המספר 12. אמנם ניתן לבצע על רשימות פעולות אריתמטיות, אבל אם תוצאת הפעולה יוצאת מטווח הערכים המוגדר עבור הרשימה, סביבת ההרצה תתייחס למשתנה הרשימה לפי הערך השלם ולא לפי הערך הליטרלי, כלומר הערך האחרון (דצמבר) הינו 11, עבור הערך השלם 12 אין ייצוג, ולכן יוצג הערך השלם 12.

14. הקש Enter כדי לסגור את התוכנית ולחזור לסביבת הפיתוח.

עבודה עם מבנים

בפרק 8 ראית שמחלקות מגדירות סוגי הפניה אשר מאוחסנים תמיד במצבור (heap). במקרים מסוימים, המחלקה עשויה להכיל כה מעט נתונים, שכמות הזיכרון המוקצה לניהולם הינה גדולה מנפח הנתונים עצמם. במקרים מסוימים עדיף להגדיר את הסוג בתור מבנה, כי מבנים מאוחסנים במחסנית וכמות הזיכרון הנדרשת לניהולם קטנה יותר (כל עוד המבנה קטן דיו).

כמו במקרה של מחלקה (ובניגוד לרשימה), יכולים להיות למבנה שדות, שיטות ובנאים משלו, אולם הם יהיו מסוג ערך ולא מסוג הפניה.

סוגי מבנה שכיחים

ייתכן שלא שמת לב, אבל השתמשת כבר במבנים בתרגילים שבפרקים הקודמים. בשפת #C, הסוגים הנומריים הפרימיטיביים **int**, **Long** ו-**float** הינם שמות המייצגים את המבנים **System.Int32**, **System.Int64** ו-**System.Single** בהתאמה. משמעות הדבר היא שלמעשה ניתן להפעיל שיטות על משתנים וליטרלים מסוגים אלה. לדוגמה, כל המבנים הבאים מספקים את השיטה **ToString** המסוגלת להמיר ערך מספרי למחרוזת המייצגת אותו. כל המשפטים הבאים הינם משפטים חוקיים בשפת #C:

```
int i = 99;
Console.WriteLine(i.ToString());
Console.WriteLine(55.ToString());
float f = 98.765F;
Console.WriteLine(f.ToString());
Console.WriteLine(98.765F.ToString());
```

הסיבה לכך שלא נתקלים לעיתים קרובות בשימוש כזה בשיטה **ToString** היא שהשיטה **Console.WriteLine** קוראת לשיטה זו באופן אוטומטי בעת הצורך. נפוץ הרבה יותר הוא השימוש בשיטות סטטיות אשר נחשפו על ידי מבנים אלה. לדוגמה, בפרקים הקודמים השיטה הסטטית **Int32.Parse** שימשה אותנו להמרת מחרוזת לערך השלם המקביל לה:

```
string s = “42”;
int i = Int32.Parse(s);
```

הערה

מכיוון ש-**int** הוא כינוי ל-**Int32**, ניתן להשתמש גם ב-**int.Parse**.

מבנים אלה כוללים שדות סטטיים אחדים שימושיים. לדוגמה, **Int32.MaxValue** הוא הערך המקסימלי שניתן לאחסן ב-**int**.

הטבלה הבאה מציגה את הסוגים הפרימיטיביים הקיימים ב-C#, את הסוגים המקבילים להם ב-NET Framework., ומציינת אם כל אחד מהם הוא מחלקה או מבנה.

Keyword	Type equivalent	Class or structure
bool	**System.Boolean**	Structure
byte	**System.Byte**	Structure
decimal	**System.Decimal**	Structure
Double	**System.Double**	Structure
Float	**System.Single**	Structure
Int	**System.Int32**	Structure
Long	**System.Int64**	Structure
Object	**System.Object**	Class
Sbyte	**System.SByte**	Structure
Short	**System.Int16**	Structure
String	**System.String**	Class
Uint	**System.UInt32**	Structure
Ulong	**System.UInt64**	Structure
Ushort	**System.UInt16**	Structure

הכרזה על סוגי מבנה

כדי להכריז על סוגי ערך מבנה משלך, עליך להשתמש במילת המפתח **struct**, לציין אחריה את שם הסוג, ולבסוף לרשום את גוף המבנה מתוחם בשני סוגריים מסולסלים. להלן מבנה בשם **Time** המכיל שלושה שדות ציבוריים מסוג **int** הנקראים **hours**, **minutes** ו-**seconds**:

```
struct Time
{
   public int hours, minutes, seconds;
}
```

בדומה למחלקות, ברוב המקרים לא מומלץ לקבוע את שדות המבנה **כציבוריים**, כי אין כל דרך להבטיח ששדות כאלה יכילו ערכים חוקיים. לדוגמה, כל אחד יוכל לקבוע את ערך **הדקות** ו**השניות** לערך הגבוה מ-60. לכן עדיף שהשדות יהיו **פרטיים** ושהמבנה יכיל בנאים ושיטות, כפי שנעשה בדוגמה הבאה:

```
struct Time
{
   public Time(int hh, int mm, int ss)
   {
      hours = hh % 24;
      minutes = mm % 60;
      seconds = ss % 60;
   }
   public int Hours()
   {
      return hours;
   }
   ...
   private int hours, minutes, seconds;
}
```

הערה

על פי ברירת המחדל, לא ניתן להפעיל חלק גדול מהאופרטורים המשותפים על מבנים שיצרת בעצמך. לדוגמה, אינך יכול להשתמש באופרטורים כגון == או =! עבור משתנים מסוג **struct** אשר יצרת בעצמך. אולם, תוכל להכריז על אופרטורים ייעודיים עבור הסוגים **struct** שיצרת וליישם אותם. בפרק 19 תלמד כיצד לעשות זאת.

במבנים כדאי להשתמש כדי ליישם רעיונות פשוטים אשר מאפיינם המרכזי הוא הערך. לדוגמה, **int** הוא סוג ערך, כי המאפיין העיקרי הוא הערך שלו. תוצאת ההעתקה של משתנה מסוג ערך היא שני עותקים של הערך. עם זאת, תוצאת ההעתקה של משתנה מסוג הפניה היא שתי הפניות אל אותו אובייקט. לסיכום, השתמש במבנים ליישום מקרים פשוטים שבהם סביר להעתיק ערכים, והשתמש במחלקות עבור מקרים מורכבים יותר שבהם לא כדאי להעתיק ערכים.

הבדלים בין מבנים למחלקות

מבנים ומחלקות דומים מאוד זה לזה מבחינה תחבירית, אולם יש מספר הבדלים חשובים ביניהם, אשר ראוי לציין. נבחן כעת חלק מההבדלים הללו:

- לא ניתן להכריז על בנאי ברירת מחדל (בנאי ללא פרמטרים) עבור מבנה. הדוגמה הבאה הייתה יכולה לעבור הידור אם **Time** היה מחלקה, אולם מכיוון ש-**Time** הוא מבנה (**struct**), לא ניתן להדר זאת.

```
struct Time
{
   public Time() { ... } // compile time error
   ...
}
```

הסיבה לכך שאינך יכול להכריז על בנאי ברירת מחדל משלך **במבנה**, היא שמהדר **תמיד** יוצר אחד. במחלקה, המהדר יוצר בנאי ברירת מחדל רק אם אינך כותב בנאי בעצמך.

בדומה למחלקה, בנאי ברירת המחדל אשר נוצר על ידי המהדר עבור מבנים, מאתחל תמיד את ערך השדות ל-0, **false** או **null**. לפיכך, עליך לוודא שערך המבנה שנוצר על ידי בנאי ברירת המחדל הגיוני, ושהבנאי פועל באופן שיש משמעות לערכים אלה. אם אינך רוצה להשתמש בערכים שנוצרו, אתחל את השדות לערכים אחרים באמצעות בנאי שאינו ברירת המחדל. אולם, אם לא תאתחל שדה בתוך הבנאי, המהדר לא יאתחל אותו עבורך. משמעות הדבר היא שעליך לאתחל באופן ברור את כל השדות בכל הבנאים שבמבנה, אחרת תתקבל שגיאה בזמן ההידור. לדוגמה, למרות שהדוגמה הבאה הייתה עוברת הידור ומאתחלת את השדה **seconds** ל-0 אם **Time** היה מחלקה, ההידור של **Time** נכשל, כי זהו מבנה.

```
struct Time
{
   public Time(int hh, int mm)
   {
      hours = hh;
      minutes = mm;
   }  // compile time error: seconds not initialized
   ...
   private int hours, minutes, seconds;
}
```

- במחלקה ניתן לאתחל את השדות בעת ההכרזה, ובמבנה לא ניתן לעשות זאת. הדוגמה הבאה יכולה הייתה לעבור הידור אם **Time** היה מחלקה, אבל מכיוון שזהו מבנה, תהיה שגיאה בשלב ההידור. כאן אפשר לחזור ולהזכיר את הכלל שכל מבנה חייב לאתחל את כל השדות שלו בכל הבנאים שלו:

```
struct Time
{
   ...
   private int hours = 0; // compile time error
   private int minutes;
   private int seconds;
}
```

הטבלה הבאה מסכמת את ההבדלים בין מבנה למחלקה:

שאלה	מבנה	מחלקה
סוג ערך או סוג הפניה?	סוג ערך.	סוג הפניה.
האם מופעים (instances) מאוחסנים במחסנית או במצבור?	מופעים נקראים **ערכים** ומאוחסנים במחסנית.	מופעים נקראים **אובייקטים** ומאוחסנים במצבור.
האם ניתן להכריז על בנאי ברירת מחדל?	לא	כן
אם תכריז על בנאי משלך, האם המהדר ייצור בכל זאת בנאי ברירת מחדל?	כן	לא
אם לא תאתחל את השדות של הבנאי שיצרת, האם המהדר יאתחל אותם עבורך באופן אוטומטי?	לא	כן
האם מותר לאתחל שדות בעת ההכרזה עליהם?	לא	כן

יש הבדלים נוספים בין מחלקות למבנים הקשורים לנושא ההורשה (inheritance). לדוגמה, מחלקה יכולה לרשת ממחלקת בסיס (base class), אולם מבנה אינו יכול לעשות זאת. הבדלים אלה מפורטים בפרק 12. כעת, לאחר שלמדת כיצד להכריז על מבנים, הצעד הבא הוא ללמוד כיצד להשתמש בהם כדי ליצור ערכים.

הכרזה על משתנה מסוג מבנה

לאחר הגדרת סוג **מבנה**, תוכל להשתמש בו בדיוק באותו אופן שבו אתה משתמש בסוגים אחרים. לדוגמה, לאחר הגדרת המבנה **Time**, תוכל ליצור משתנים, שדות ופרמטרים מהסוג **Time**, כמודגם להלן:

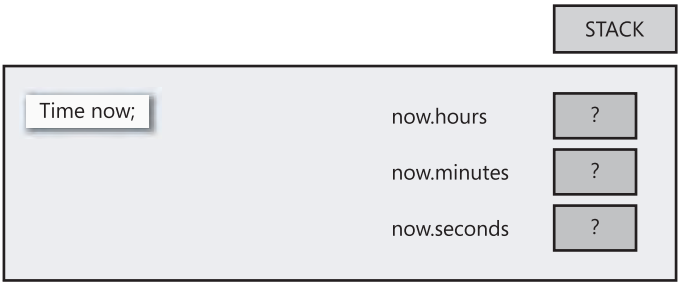
```
struct Time
{
   ...
   private int hours, minutes, seconds;
}
class Example
{
   public void Method(Time parameter)
   {
      Time localVariable;
      ...
   }
   private Time currentTime;
}
```

אתחול מבנים

קודם לכן תיארנו כיצד שדות ב**מבנה** מאותחלים באמצעות בנאי. אולם מכיוון שמבנים הם מסוג ערך, ניתן ליצור משתנים מסוג **מבנה** מבלי לקרוא לבנאי, כמודגם להלן:

```
Time now;
```

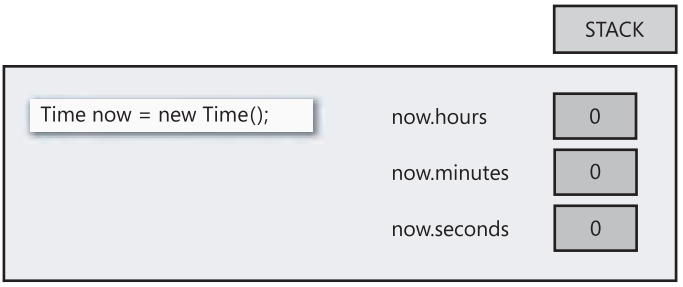
בדוגמה זו המשתנה נוצר, אולם שדותיו נותרים בלתי מאותחלים. כל ניסיון לגשת לערכים בשדות אלה יגרום לשגיאת הידור. התרשים הבא ממחיש את מצב השדות במשתנה **now**:



אם תקרא לבנאי, החוקים השונים שתוארו קודם ואשר תקפים לגבי בנאים של מבנים, יבטיחו שכל השדות במבנה יאותחלו:

```
Time now = new Time();
```

הפעם, בנאי ברירת המחדל יאתחל את השדות במבנה כפי שמוצג בתרשים הבא:



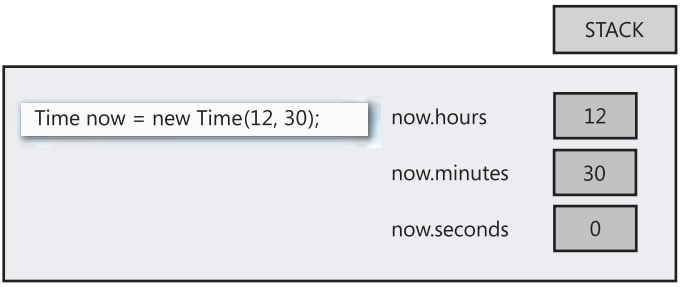
שים לב שבשני המקרים, המשתנה **Time** נוצר במחסנית. אם יצרת בנאי מבנה משלך, תוכל להשתמש בו גם כדי לאתחל משתנה מסוג **מבנה**. זכור שבנאי מבנה חייב לאתחל את כל השדות שלו. לדוגמה:

```
struct Time
{
   public Time(int hh, int mm)
   {
      hours = hh;
      minutes = mm;
      seconds = 0;
   }
   ...
   private int hours, minutes, seconds;
}
```

בדוגמה הבאה **now** מאותחל על ידי קריאה לבנאי שהוגדר על ידי המשתמש:

```
Time now = new Time(12, 30);
```

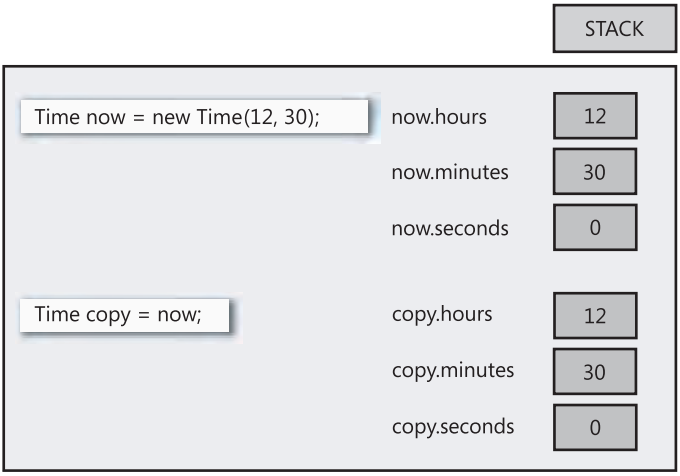
התרשים הבא ממחיש את תוצאות הדוגמה שלעיל:



העתקת משתני מבנה

ניתן לאתחל משתנה **struct** באמצעות משתנה **struct** אחר, או להציב משתנה **struct** במשתנה **struct**, אבל רק אם המשתנה **struct** שמימין מאותחל לחלוטין (כלומר, כל שדותיו מאותחלים). למשל, ניתן להדר את הקוד הזה, כי **now** מאותחל לחלוטין. התרשים גם מציג את תוצאות ביצוע ההצבה:

```
Time now = new Time(12, 30);
Time copy = now;
```



הידור הקוד הבא ייכשל, כי **now** אינו מאותחל:

```
Time now;
Time copy = now; // compile time error: now unassigned
```

בעת העתקת משתנה **struct**, כל שדה שנמצא מימין מועתק באופן מדויק אל השדה המקביל לו בצד שמאל. העתקה מבוצעת במהירות ובבלוק אחד ולעולם אינה זורקת חריג. נדגיש שוב שסוג הצבה זה גורם להעתקת המבנה כולו. השווה התנהגות זו לפעולה שהייתה מתבצעת אם **Time** היה מחלקה. במקרה זה, שני המשתנים (**now** ו-**copy**) היו מפנים **לאותו** האובייקט במצבור.

הערה

מתכנתים מרקע תכנות של C++ צריכים לשים לב שלא ניתן לשנות את אופן ההעתקה.

כעת הגיע הזמן ליישם את הידע שלמדת. בתרגיל הבא עליך ליצור סוג **מבנה** המייצג תאריך ולהשתמש בו.

צור סוג מבנה והשתמש בו

1. פתח את הפרויקט **StructsAndEnums** שבתיקייה My Documents\Microsoft Press\Visual CSharp Step by Step\Chapter 9\StructsAndEnums.
2. הצג את קובץ המקור Date.cs בחלון Code and Text Editor.
3. הוסף **struct** בשם **Date** אל מרחב השמות **StructsAndEnums**.

המבנה צריך להכיל שלושה שדות פרטיים (private): אחד בשם **year** מסוג **int**, אחד בשם **month** מסוג **Month** (כפי שהכרזנו בתרגיל הקודם), ואחד בשם **day** מסוג **int**. על המבנה **Date** להיראות כעת כך:

```
struct Date
{
   private int year;
   private Month month;
   private int day;
}
```

כעת נסה לחשוב כיצד ייראה בנאי ברירת המחדל אשר ייצור המהדר עבור **Date**. בנאי זה יאתחל את **השנה** ל-0, את **החודש** ל-0 (הערך של ינואר) ואת **היום** ל-0. הערך 0 אינו חוקי עבור **השנה** (לא הייתה שנת 0), וגם לא עבור **היום** (כל חודש מתחיל ב-1). אחת הדרכים לפתור את הבעיה היא לתרגם את הערכים של **השנה** **והיום**. כדי לעשות זאת עליך ליישם את המבנה **Date** באופן שכאשר השדה **year** מכיל את הערך **Y**, ערך זה ייצג בעצם את השנה 1900+**Y**, וכאשר השדה **day** יכיל את הערך **D**, אותו הערך ייצג את היום 1+**D**. לפיכך, בנאי ברירת המחדל יאתחל את השדות לערכים המייצגים את **התאריך** 1 בינואר 1900.

4. הוסף בנאי **ציבורי** למבנה **Date**. בנאי זה אמור לקבל שלושה פרמטרים: **int** בשם **ccyy** עבור השנה, **Month** בשם **mm** עבור **החודש** ו-**int** בשם **dd** עבור **היום**. השתמש בשלושת הפרמטרים הללו כדי לאתחל את השדות המקבילים להם. שדה **year** המכיל את הערך **Y** מייצג את השנה 1900+**Y**, לכן עליך לאתחל את השדה **year** לערך **ccyy**-1900. שדה **day** המכיל את הערך **D** מייצג את היום 1+**D**, לכן עליך לאתחל את השדה **day** לערך **dd**-1.

המבנה **Date** נראה כעת כך:

```
struct Date
{
   public Date(int ccyy, Month mm, int dd)
   {
      this.year = ccyy - 1900;
      this.month = mm;
      this.day = dd - 1;
   }
   private int year;
   private Month month;
   private int day;
}
```

5. הוסף שיטה **ציבורית ToString** למבנה **Date**, לאחר הסוגריים המסולסלים. שיטה זו אינה מקבלת ארגומנטים ומחזירה מחרוזת המייצגת את התאריך. זכור שערך השדה **year** מייצג את השנה 1900+**year**, וערך השדה **day** מייצג את היום 1+**day**.

הערה

השיטה **ToString** שונה במקצת מהשיטות שראית עד כה. כל סוג, כולל מבנים ושיטות אשר הגדרת בעצמך, כולל בתוכו תמיד את השיטה **ToString**. התנהגות ברירת מחדל של השיטה היא המרה של ערך הנתון במשתנה אל מחרוזת המייצגת את אותו מידע. לעיתים התנהגות ברירת המחדל היא בעלת משמעות, ולעיתים היא לא כל-כך משמעותית. לדוגמה, לפי ברירת המחדל של השיטה **ToString**, היא יוצרת את המחרוזת "StructsAndEnum.Date". עליך להגדיר גרסה חדשה לשיטה זו אשר תעקוף את התנהגות ברירת המחדל באמצעות מילת המפתח **override**. נדון בהרחבה על שיטות עוקפות בפרק 12.

השיטה **ToString** נראית כך:

```
public override string ToString()
{
  return this.month + " " + (this.day + 1) + " " + (this.year +  1900);
}
```

הערה

סימני הפלוס (+) שבתוך הסוגריים הינם אופרטורים המבצעים חיבור. סימני הפלוס האחרים הינם אופרטורים לשרשור מחרוזות. ללא הסוגריים, כל סימני הפלוס היו נחשבים לאופרטורים לשרשור מחרוזות, כי הביטוי הנבדק הינו מחרוזת. חשוב לשים לב כאשר אותו הסימן מייצג אופרטור מסוגים שונים באותו הביטוי.

6. הצג את קובץ המקור Program.cs בחלון Code and Text Editor.

7. הוסף משפט בסוף השיטה **Entrance** שתפקידו להכריז על משתנה מקומי **defaultDate** ולאתחל אותו לערך מסוג **Date** הנוצר כתוצאה מהפעלת הבנאי **defaultDate**. הוסף משפט נוסף לשיטה **Entrance** אשר מדפיסה את **defaultDate** על המסך באמצעות קריאה לשיטה **Console.WriteLine**.

הערה

השיטה **Console.WriteLine** קוראת באופן אוטומטי לשיטה **ToString** של הארגומנט שלה, כדי להמיר את הארגומנט למחרוזת.

השיטה **Entrance** נראית כעת כך:

```
static void Entrance()
{
   ...
   Date defaultDate = new Date();
   Console.WriteLine(defaultDate);
}
```

8. בתפריט Debug בחר Start Without Debugging כדי לבנות ולהפעיל את התוכנית. ודא שעל המסך מופיע התאריך January 1 1900 (לפני כן יוצג הפלט המקורי של השיטה **Entrance**).

9. הקש Enter כדי לחזור לסביבת הפיתוח.

10. בחלון Code and Text Editor חזור לשיטה **Entrance** והוסף שני משפטים. המשפט הראשון מכריז על משתנה מקומי **halloween** (ליל כל הקדושים) ומאתחל אותו ל-October 31 2005. המשפט השני מדפיס את הערך של **halloween** על המסך.

השיטה **Entrance** נראית כעת כך:

```
static void Entrance()
{
   ...
   Date halloween = new Date(2005, Month.October, 31);
   Console.WriteLine(halloween);
}
```

הערה

בעת הקלדת מילת המפתח **new**, אחריו Date וסוגר ימני, הרכיב Intellisense יזהה באופן אוטומטי שיש שני בנאים נוספים הזמינים עבור הסוג **Date**.

11. בתפריט Debug בחר Start Without Debugging. ודא שהתאריך October 31 2005 מופיע על המסך לאחר המידע הקודם.

12. הקש Enter כדי לסגור את התוכנית.

השתמשת בהצלחה במילות המפתח **enum** ו-**struct** כדי להכריז על סוגי ערך (value types) משלך ולהשתמש בסוגים אלו כחלק מקוד.

☯ **אם ברצונך להמשיך לפרק הבא**

השאר את Visual Studio 2005 פעילה ועבור לפרק 10.

☯ **אם ברצונך לכבות כעת את Visual Studio 2005**

פתח את תפריט Menu ולחץ Exit. במידה ומופיעה תיבת דו-שיח Save, לחץ Yes.

פרק 9 – טבלה מסכמת

המשימה	צריך
להכריז על סוג רשימה (enumeration type).	להקליד את מילת המפתח **enum**, אחריה שם הסוג ואחריו זוג סוגריים מסולסלים המכילים את שמות הליטרלים ברשימה אשר מופרדים זה מזה בפסיקים. לדוגמה: `enum Season {Spring, Summer, Fall, Winter }`
להכריז על משתנה רשימה.	להקליד את השם של סוג הרשימה, אחריו שם המשתנה ולבסוף נקודה פסיק (;). לדוגמה: `Season currentSeason;`

לאתחל משתנה רשימה או להציב בו ערך.	לכתוב שם של ליטרל השייך לרשימה בשילוב עם שם סוג הרשימה שהוא משתייך אליה. לדוגמה:
	`currentSeason = Spring; // compile time error`
	`currentSeason = Season.Spring; // okay`

להכריז על סוג מבנה (**struct** type).	להקליד את מילת המפתח **struct**, אחריה שם סוג המבנה ואחריו גוף **המבנה** (הבנאים, השיטות והשדות). לדוגמה:
	`struct Time`
	`{`
	`  public Time(int hh, int mm, int ss)`
	`  { ... }`
	`  ...`
	`  private int hours, minutes, seconds;`
	`}`

להכריז על משתנה **struct**.	להקליד את השם של סוג המבנה, אחריו שם המשתנה ולבסוף נקודה פסיק (;). לדוגמה: `Time now;`

לאתחל משתנה מבנה או להציב בו ערך.	להציב במשתנה את הערך **struct** הנוצר כתוצאה מהקריאה לבנאי של המבנה. לדוגמה:
	`Time lunch = new Time(12, 30, 0);`
	`lunch = new Time(12, 30, 0);`

פרק 10

מערכים ואוספים

בסיום פרק זה, תוכל:

- ☯ להכריז, לאתחל, להעתיק ולהשתמש במשתנים מסוג **מערך** (Array).
- ☯ להכריז, לאתחל, להעתיק ולהשתמש במשתנים מסוגי **אוסף** (Collection) שונים.

עד כה למדת ליצור ולהשתמש בסוגים רבים של משתנים. אולם לכל סוגי המשתנים שפגשת עד כה יש דבר אחד משותף - הם משמשים לאחסון מידע של **רכיב** (element) אחד בלבד (**int**, **float**, **Circle**, **Time** וכדומה). מה קורה כאשר יש צורך לפעול על קבוצה של רכיבים? אחד הפתרונות הוא ליצור משתנה עבור כל רכיב בקבוצה, אך פתרון זה עלול לגרום מספר בעיות נוספות: כמה משתנים נחוצים בכלל? מה יהיו שמותיהם? כאשר נרצה לבצע פעולה זהה על כל אחד מהרכיבים (כמו הגדלה באחד של כל מספר בקבוצה של שלמים), כיצד נימנע מלחזור על אותו הקוד מספר רב של פעמים? ובנוסף, פתרון זה מתבסס על ההנחה שאתה יודע כמה פרטים ישנם בזמן כתיבת התוכנית, אולם מה הסבירות שבאמת תדע זאת? לדוגמה, נניח שאתה כותב יישום בנקאי אשר מקבל מספרי חשבון שחרגו ממסגרת האשראי ושולח התראה לבעלי החשבון ממאגר נתונים מרכזי של הבנק, כיצד ניתן לדעת כמה מספרי חשבון יש במאגר?

מערכים (arrays) ואוספים (collections) מספקים מנגנון המשמש לפתרון הבעיות שמציבות השאלות הללו.

מהו מערך?

מערך (**array**) הוא רצף של אלמנטים. כל האלמנטים ברצף חייבים להיות מאותו סוג נתונים (בניגוד לשדות **במבנה** (struct) או **במחלקה** (class), שיכולים להיות מסוגים שונים). האלמנטים במערך מאוחסנים בבלוק זיכרון רציף, ומתויקים במחשב באמצעות אינדקס המבוסס על מספר שלם, וכך קל לטפל בהם ולנווט ביניהם.

הכרזה על מערכים

כדי להכריז על משתנה מסוג מערך, עליך לציין את שם סוג הנתונים שיאוחסנו בו, אחריו צמד סוגריים מרובעים ולבסוף לציין את שם המשתנה. הסוגריים המרובעים מציינים שהמשתנה הוא מערך. לדוגמה, כדי להכריז על מערך של משתנים pins מסוג decimal, עליך לכתוב:

```
int[] pins; // Personal Identification Numbers
```

מתכנתים מרקע תכנות של Microsoft Visual Basic צריכים לשים לב לשימוש בסוגריים המרובעים (ולא בסוגריים רגילים). מתכנתים מרקע של C ו- C++ צריכים לשים לב שגודל המערך אינו כלול בהכרזה. מתכנתים מרקע של Java צריכים לשים לב שעליהם לכתוב את הסוגריים המרובעים **לפני** שם המשתנה.

הערה

סוג הרכיבים במערך אינו מוגבל לסוגי נתונים פרימיטיביים. ניתן ליצור מערכים של מבנים, רשימות ומחלקות. לדוגמה, כדי ליצור מערך של מבני **Time** כתוב את המשפט Time[] times;.

טיפ

רצוי להקצות למערכים שמות ברבים, כגון places (כל אחד מהרכיבים הוא יחידת **place** אחת), **Persons** (כל אחד מהרכיבים הוא יחידת **Person** אחת) או **times** (כל אחד מהרכיבים הוא יחידת **time** אחת).

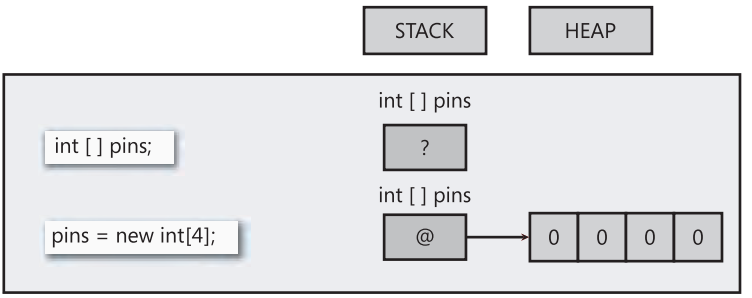
יצירת מופעים של מערכים

מערכים הם מסוג **הפניה** (reference), ללא תלות בסוג הרכיבים שלהם. משמעות הדבר היא שמשתנה מערך מפנה **למופע** (instance) של המערך במצבור, באותו אופן שמשתנה מסוג מחלקה מפנה לאובייקט במצבור, ואינו מאחסן את הרכיבים ישירות במחסנית, בדומה למבנה (כדי לחזור על ערכים והפניות ועל ההבדלים בין מחסנית למצבור, עיין בפרק 8). זכור שבעת הכרזה על משתנה מחלקה, הזיכרון עבור האובייקט אינו מוקצה עד שתיצור מופע באמצעות מילת המפתח **new** (חדש). הקצאת הזיכרון למערכים נוהגת באותו אופן - בעת הכרזה על משתנה מערך אינך קובע את גודלו. עליך לציין את גודל המערך רק כאשר אתה יוצר את המופע שלו.

כדי ליצור מופע של מערך, עליך להשתמש במילת המפתח **new** ולאחריה שם סוג הרכיב, ולבסוף גודל המערך שאתה יוצר ייכתב בין סוגריים מרובעים. יצירת מערך גם גורמת לאתחול רכיביו לערכי ברירת המחדל המוכרים 0, **null** או **false**, בהתאם לסוג. לדוגמה, כדי ליצור ולאתחל מערך חדש המורכב מארבעה מספרים שלמים עבור המשתנה **pins** שהכרזנו עליו קודם לכן, יש לכתוב את המשפט הבא:

```
pins = new int[4];
```

התרשים הבא ממחיש את תוצאות ביצוע המשפט:



גודל המופע של המערך אינו חייב להיות קבוע. ניתן לחשבו בזמן הפעלת התוכנית, כמוצג בדוגמה הבאה:

```
int size = int.Parse(Console.ReadLine());
int[] pins = new int[size];
```

ניתן ליצור מערכים רב-ממדיים. לדוגמה, כדי ליצור מערך דו-ממדי, עליו לכלול שני אינדקסים של מספרים שלמים, אחד לכל ממד. לא נרחיב בספר זה על מערכים דו-ממדיים, אולם הנה דוגמה למערך כזה:

```
int[,] table = new int[4,6];
```

אתחול משתנים מסוג מערך

בעת יצירת מופע של מערך, כל הרכיבים בו מאותחלים לערך ברירת מחדל התלוי בסוג הרכיב. באפשרותך לשנות את ההתנהגות הזו, ולאתחל את הרכיבים במערך לערכים שתבחר בעצמך. כדי לעשות זאת, עליך להקליד רשימת ערכים המופרדים זה מזה בפסיקים ומתוחמים על ידי צמד סוגריים מסולסלים. לדוגמה, כדי לאתחל את **pins** כמערך המכיל 4 משתנים שערכיהם 9, 3, 7, ו-2, עליך לכתוב:

```
int[] pins = new int[4]{ 9, 3, 7, 2 };
```

הערכים בין הסוגריים המסולסלים אינם חייבים להיות קבועים. הם יכולים להיות ערכים המחושבים בזמן הפעלת התוכנית, כמודגם להלן:

```
Random r = new Random();
int[] pins = new int[4]{ r.Next() % 10, r.Next() % 10,

                         r.Next() % 10, r.Next() % 10 };
```

הערה

המחלקה **System.Random** היא מחוללת מספרים פסבדו-אקראיים. השיטה **Next** מחזירה מספר אקראי שאינו שלילי.

מספר הערכים בין הסוגריים המסולסלים חייב לתאום בדיוק לגודל המופע של המערך:

```
int[] pins = new int[3]{ 9, 3, 7, 2 }; // compile time error
int[] pins = new int[4]{ 9, 3, 7 }; // compile time error
int[] pins = new int[4]{ 9, 3, 7, 2 }; // okay
```

בעת אתחול של משתנה מערך, ניתן למעשה להשמיט את מילת הביטוי **new** ואת גודל המערך. המהדר ייחשב את גודל המערך על פי מספר הנתונים, וייצור את הקוד ליצירת המערך. לדוגמה:

```
int[] pins = { 9, 3, 7, 2 };
```

אם אתה יוצר מערך של מבנים (structs), תוכל לאתחל כל מבנה במערך באמצעות קריאה לבנאים של המבנים, כמודגם להלן:

```
Time[] schedule = { new Time(12,30), new Time(5,30) };
```

גישה לרכיבים יחידים במערך

כדי לגשת לרכיב מסוים במערך, עליך לציין מספר אינדקס המפנה לערך הנחוץ לך. לדוגמה, כדי להציב את תוכן רכיב 2 של מערך **Pins** במשתנה מסוג **int**, עליך להשתמש בקוד הבא:

```
int myPin;
myPin = pins[2];
```

ניתן גם לשנות ערך פריט במערך, על ידי הצבת ערך במספר הרכיב הרצוי:

```
myPin = 1645;
pins[2] = myPin;
```

אינדקסים של מערכים מתחילים ב-0. הרכיב הראשון במערך מאוחסן באינדקס 0 ולא באינדקס 1. ערך אינדקס 1 מכיל את הרכיב השני.

הגישה לרכיבי המערך נבדקת על ידי המהדר, ואם תשתמש במספר אינדקס קטן מאפס או גדול או שווה לאורך המערך, המהדר יזרוק חריג **IndexOutOfRangeException**, כפי שניתן לראות בדוגמה הבאה:

```
try
{
   int[] pins = { 9, 3, 7, 2 };
   Console.WriteLine(pins[4]); // error, the 4th element is at
   index 3
}
catch (IndexOutOfRangeException ex)
{
   ...
}
```

דפדוף במערך

למערכים יש מספר מאפיינים ושיטות מובנים ושימושיים ביותר (שיטות ומאפיינים אלה מועברים בהורשה לכל המערכים מהמחלקה **System.Array** שנמצאת ב-NET Framework. של Microsoft). השתמש במאפיין **Length** כדי לדעת כמה רכיבים יש במערך.

ניתן להשתמש במאפיין **Length** בשילוב עם משפט **for** כדי לדפדף (iterate) בין כל רכיבי המערך. הקוד הבא מציג במסך את ערכי כל הרכיבים שבמערך **pins**:

```
int[] pins = { 9, 3, 7, 2 };
for (int index = 0; index != pins.Length; index++)
{
   int pin = pins[index];
   Console.WriteLine(pin);
}
```

הערה

Length הוא מאפיין ולא שיטה, ולכן אין צורך להשתמש בסוגריים כאשר קוראים לו. על מאפיינים נלמד בפרק 14.

מתכנתים לא מנוסים שוכחים לעיתים קרובות שאינדקס של המערך מתחיל באפס, ולכן הרכיב האחרון במערך הוא **Length** – 1. שפת C# כוללת משפט **foreach** המשמש לדפדוף בין הרכיבים השונים במערך מבלי לדאוג לסוגיות כאלו. הנה לדוגמה לולאת **for** הקודמת, אולם הפעם בגרסת **foreach**:

```
int[] pins = { 9, 3, 7, 2 };
foreach (int pin in pins)
{
   Console.WriteLine(pin);
}
```

המשפט **foreach** מכריז על משתנה איטרציה (בדוגמה זו **int pin**) המקבל באופן אוטומטי את ערך כל אחד מרכיבי המערך. מבנה זה הרבה יותר ברור, ומביע את הפעולה שהקוד מבצע באופן הרבה יותר ברור, מובן וישיר מאשר לולאת **for** המורכבת. לרוב עדיף להשתמש במשפט **foreach** כדי לדפדף במערכים, אולם במספר מצבים יהיה עליך להשתמש בכל זאת במשפט **for**:

- משפט **foreach** ידפדף תמיד במערך כולו. כאשר אתה רוצה לדפדף רק בחלק מסוים של המערך (בחצי הראשון שלו למשל) או לדלג על חלק מהרכיבים (על כל רכיב שלישי, למשל), עדיף להשתמש במשפט **for**.
- משפט **foreach** מתחיל לדפדף תמיד מאינדקס 0 ומגיע עד לאינדקס **length** – 1. כאשר ברצונך לדפדף לאחור, עדיף להשתמש במשפט **for**.
- אם קיים צורך לדעת בגוף הלולאה את מספר האינדקס ולא רק את ערך הרכיב הסודר, תהיה חייב להשתמש במשפט **for**.
- כדי לערוך את רכיבי המערך, עליך להשתמש במשפט **for**. הסיבה לכך היא שמשתנה האיטרציה במשפט **foreach** הוא עותק לקריאה בלבד של כל רכיב במערך.

העתקת מערכים

מערכים הם מסוג הפניה. משתנה מערך מכיל הפניה למופע של המערך. משמעות הדבר היא, שתוצאת ההעתקה של משתני מערך תהיה שני משתנים המפנים אל אותו מופע של המערך, לדוגמה:

```
int[] pins = { 9, 3, 7, 2 };
int[] alias = pins; // alias and pins refer to the same array
instance
```

בדוגמה זו, שינוי הערך של pin[1] יתבטא גם ב- alias[1].

כדי ליצור עותק של **מופע המערך** (הנתונים המאוחסנים במצבור) שהמשתנה מערך מפנה אליו, עליך לבצע שתי פעולות. תחילה עליך ליצור מופע חדש מאותו סוג ובאותו גודל של המערך שברצונך להעתיק. לדוגמה:

```
int[] pins = { 9, 3, 7, 2 };
int[] copy = new int[4];
```

קוד זה יפעל, אמנם, אולם כאשר תרצה לשנות מאוחר יותר את גודל המערך המקורי, תצטרך לזכור לשנות גם את גודל העותק שלו. לכן עדיף לקבוע את גודל העותק באמצעות המאפיין **Length**, כמודגם להלן:

```
int[] pins = { 9, 3, 7, 2 };
int[] copy = new int[pins.Length];
```

כעת הערכים שבמשתנה המערך copy מאותחלים לערך ברירת המחדל 0.

הפעולה השנייה שעליך לבצע היא להציב ברכיבי המערך החדש ערכים זהים לאלה של המערך המקורי. ניתן לעשות זאת באמצעות משפט **for**, כמודגם להלן:

```
int[] pins = { 9, 3, 7, 2 };
int[] copy = new int[pins.Length];
for (int i = 0; i != copy.Length; i++)
{
   copy[i] = pins[i];
}
```

העתקת מערכים הינה פעולה שכיחה למדי, ולכן המחלקה **System.Array** כוללת מספר שיטות שימושיות להעתקת מערכים, אשר חוסכות את כתיבת הקוד. לדוגמה, השיטה **CopyTo** מעתיקה תכולת מערך אחד אל מערך אחר החל מאינדקס נתון:

```
int[] pins = { 9, 3, 7, 2 };
int[] copy = new int[pins.Length];
pins.CopyTo(copy, 0);
```

דרך נוספת להעתקת מערכים היא באמצעות השיטה הסטטית Copy מהמחלקה **System.Array**. בדומה לשיטה **CopyTo**, יש לאתחל את מערך היעד לפני הקריאה לשיטה.

```
int[] pins = { 9, 3, 7, 2 };
int[] copy = new int[pins.Length];
Array.Copy(pins, copy, copy.Length);
```

חלופה נוספת היא שימוש בשיטת המופע בשם **Clone**, ששייכת גם היא למחלקה **System.Array**. היא מאפשרת יצירת מערך שלם חדש והעתקתו בפעולה אחת.

```
int[] pins = { 9, 3, 7, 2 };
int[] copy = (int[])pins.Clone();
```

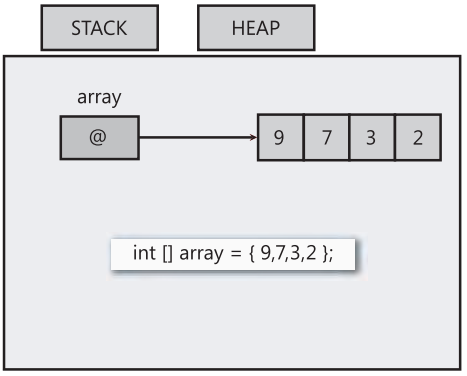
הערה

השיטה **Clone** מחזירה למעשה **אובייקט**, ולכן עליך להשליך (cast) אותה על מערך מסוג מתאים כאשר אתה מפעיל אותה. בנוסף לכך, כל שלוש השיטות הללו יוצרות עותק **שטחי** (**shallow**) של המערך. כלומר, אם המערך המועתק מכיל סוגי הפניות, השיטות יעתיקו את ההפניות האלו ולא את האובייקט אשר אליו נעשתה ההפניה. בסיום ההעתקה, שני המערכים יפנו אל אותם אובייקטים.

מהן מחלקות אוסף?

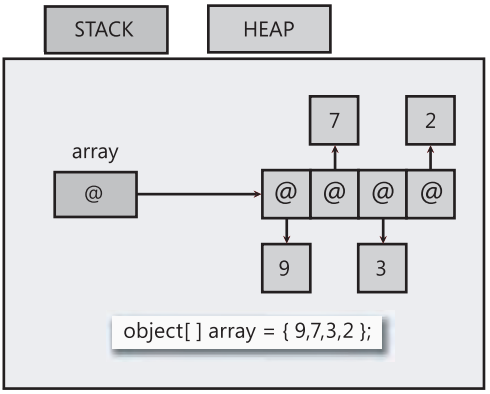
מערכים הם כלי שימושי בתכנות, אולם יש להם מגבלות. הם מהווים רק דרך אחת לאיסוף של רכיבים (elements) מאותו הסוג. מסגרת העבודה NET Framework. כוללת מספר מחלקות אשר יכולות לאסוף יחדיו רכיבים בדרכים משלהן. אלו הן **מחלקות אוסף** (**Collection classes**), והן מהוות חלק ממרחב השמות **System.Collections** ומתת-המרחבים המשתייכים אליו.

מחלקות האוסף הבסיסיות מקבלות, מחזיקות ומחזירות את הרכיבים שלהן בתור **object** (**אובייקט, עצם**). כלומר, סוג הרכיב של מחלקת אוסף יהיה תמיד **object**. כדי להבין טוב יותר את ההשלכות הנובעות מעובדה זו, עלינו להשוות בין מערך של משתני **int** (**int** הוא סוג ערך) עם מערך של אובייקטים (**object** הוא סוג הפניה). מכיוון ש-**int** הוא סוג ערך, מערך של משתני int מאחסן את המשתנים שבו באופן ישיר, כפי שמוצג בתרשים הבא:



כעת תוכל לשער מה תהיינה ההשלכות אם רכיבי המערך יהיו מסוג object. עדיין יהיה ניתן להוסיף ערכים שלמים למערך (למעשה יהיה ניתן להוסיף לו ערכים מכל סוג).

בעת הוספה של ערך שלם, הוא **נארז** (boxed) אוטומטית, והרכיב במערך (הפניה לאובייקט) מפנה לעותק הארוז של הערך השלם (נושא האריזה מוסבר בפרק 8). תהליך זה מומחש בתרשים הבא:



זכור שסוג הרכיב של מחלקת אוסף הוא **object**. המשמעות שכל משתנה מסוג ערך שתכניס לאוסף ייארז. בכל פעם שתמשוך ערך מאוסף, תצטרך **לפרוק** (unbox) אותו באמצעות **השלכה** (cast). בסעיפים הבאים ניתנת סקירה מהירה של חמשת סוגי מחלקות **אוסף** השימושיות ביותר. פרטים נוספים על כל אחת מהמחלקות ניתן למצוא בתיעוד של NET.

הערה

למעשה יש מחלקות **אוסף** אשר סוג הרכיב שלהן אינו תמיד **object**, ואשר יכולות לאחסן גם סוגי ערך בנוסף לסוגי הפניה, אולם עליך להכיר קצת יותר את שפת #C לפני שנוכל לדון בהם. במחלקות **אוסף** נעסוק בפרק 17.

המחלקה ArrayList

המחלקה **ArrayList** שימושית לצורך ערבוב רכיבים במערך. יש מספר מקרים שבהם מערך רגיל עשוי להגביל במידה רבה מדי:

- כדי לשנות את גודל המערך, עליך ליצור מערך חדש, להעתיק את הרכיבים (או את חלקם אם המערך החדש קטן יותר) ולבסוף - לעדכן את ההפניות של המערך.
- כדי להסיר רכיב מהמערך, עליך ליצור עותק של הרכיב, ולאחר מכן להזיז את כל הרכיבים שמתחתיו מקום אחד כלפי מעלה (לכיוון ההתחלה). שיטה זו אינה מושלמת כי בסופו של דבר יהיו לך שני עותקים של הרכיב האחרון.
- כדי להוסיף רכיב לאמצע המערך, עליך לדחוף את כל המשתנים מקום אחד כלפי מטה (לכיוון הסוף) כדי לפנות מקום, ובסופו של דבר אתה עלול לאבד את הרכיב האחרון.
- ניתן להסיר רכיב מ-**ArrayList** באמצעות השיטה המובנית **Remove**. השיטה **ArrayList** תסדר באופן אוטומטי את הרכיבים השונים.

הערה

לא ניתן להשתמש בשיטה **Remove** בגוף לולאה **foreach** המדפדפת ב-**ArrayList**.

- ניתן להוסיף רכיב לסוף **ArrayList** באמצעות השיטה **Add**. עליך לציין את הרכיב שברצונך להוסיף. הגודל של **ArrayList** ישתנה במידת הצורך.
- ניתן להוסיף רכיב באמצע **ArrayList** באמצעות השיטה **Insert**. גם הפעם הגודל של **ArrayList** ישתנה במידת הצורך.

הדוגמה הבאה מציגה כיצד ניתן ליצור, לערוך, ולדפדף בתוכן של **ArrayList**:

```
using System;
using System.Collections;
 ...
ArrayList numbers = new ArrayList();
 ...
// fill the ArrayList
foreach (int number in new int[12]{10,9,8,7,7,6,5,10,4,3,2,1})
{
   numbers.Add(number);
}
 ...
// remove first element whose value is 7 (the 4th element, index 3)
numbers.Remove(7);
// remove the element that's now the 7th element, index 6 (10)
numbers.RemoveAt(6);
 ...
// iterate remaining 10 elements using a for statement
for (int i = 0; i != numbers.Count; i++)
{
   int number = (int)numbers[i]; // Notice the cast
   Console.WriteLine(number);
}
 ...
// iterate remaining 10 using a foreach statement
foreach (int number in numbers) // No cast needed
{
   Console.WriteLine(number);
}
```

קוד זה יפיק את הפלט הבא:

```
10
9
8
7
6
5
4
3
2
1
10
9
8
```

```
7
6
5
4
3
2
1
```

הערה

כדי למצוא את מספר הרכיבים שנמצאים באוסף עליך להשתמש במאפיין **Count**, ולא במאפיין **Length**, המשמש למציאת מספר הרכיבים במערך.

המחלקה Queue

המחלקה **Queue** מיישמת את מנגנון FIFO (First In First Out, ראשון נכנס ראשון יוצא). כל רכיב מתווסף לתור מסופו (פעולת enqueue) ויוצא מהתור מלפנים, כלומר מראשו (פעולת dequeue).

הנה דוגמה לתור ושימוש בו:

```
using System;
using System.Collections;
...
Queue numbers = new Queue();
...
// fill the queue
foreach (int number in new int[4]{9, 3, 7, 2})
{
   numbers.Enqueue(number);
   Console.WriteLine(number + " has joined the queue");
}
...
// iterate through the queue
foreach (int number in numbers)
{
   Console.WriteLine(number);
}
...
// empty the queue
while (numbers.Count != 0)
{
   int number = (int)numbers.Dequeue();
   Console.WriteLine(number + " has left the queue");
}
```

קוד זה יפיק את הפלט הבא:

```
9 has joined the queue
3 has joined the queue
7 has joined the queue
2 has joined the queue
9
3
7
2
9 has left the queue
3 has left the queue
7 has left the queue
2 has left the queue
```

המחלקה Stack

המחלקה **Stack** (**מחסנית**) מיישמת את מנגנון LIFO (Last In First Out, אחרון נכנס ראשון יוצא). כל רכיב מתווסף למחסנית בראש, מלמעלה (פעולת **push**) וגם יוצא מהמחסנית מלמעלה (פעולת **pop**). כדי לדמיין זאת, תוכל להניח שלפניך ערמת צלחות. צלחות חדשות מונחות בראש הערמה, משם הן גם נלקחות. כלומר, הצלחת האחרונה אשר הונחה על הערמה תהיה גם הראשונה להילקח (מכאן נובע שהצלחות שבתחתית הערמה נמצאות בשימוש רק לעיתים נדירות). הנה דוגמה:

```
using System;
using System.Collections;
...
Stack numbers = new Stack();
...
// fill the stack
foreach (int number in new int[4]{9, 3, 7, 2})
{
   numbers.Push(number);
   Console.WriteLine(number + “ has been pushed on the stack”);
}
...
// iterate through the stack
foreach (int number in numbers)
{
   Console.WriteLine(number);
}
...
// empty the stack
while (numbers.Count != 0)
{
   int number = (int)numbers.Pop();
   Console.WriteLine(number + “has been popped off the stack”);
}
```

קוד זה יפיק את הפלט הבא:

```
9 has been pushed on the stack
3 has been pushed on the stack
7 has been pushed on the stack
2 has been pushed on the stack
2
7
3
9
2 has been popped off the stack
7 has been popped off the stack
3 has been popped off the stack
9 has been popped off the stack
```

המחלקה Hashtable

הסוגים מערך ו-**ArrayList** מציגים דרך למצוא רכיב על-פי אינדקס של מספרים שלמים. עליך לציין מספר אינדקס בתוך סוגריים מרובעים (לדוגמה, [4]) כדי לקבל את הרכיב שבאינדקס 4 (שהוא למעשה הרכיב החמישי). אולם, לעיתים תרצה למפות את הרכיבים באמצעות אינדקס המבוסס על סוג שאינו מספר שלם, כגון **string**, **double** או **Timer**. בשפות תכנות אחרות, הדבר מכונה **associative array**. המחלקה **Hashtable** מאפשרת לבצע זאת באמצעות שני מערכי **object** פנימיים, אחד עבור **המפתחות (keys)** המשמשים למיפוי, והשני - עבור **הערכים** שאתה ממפה. כאשר אתה מכניס צמד של מפתח וערך אל **Hashtable**, המחלקה רושמת איזה מפתח מתאים לכל ערך וכך היא מאפשרת לגשת לערך באמצעות המפתח שלו. למנגנון זה יש מספר השלכות חשובות אשר כדאי להכיר:

- **Hashtable** אינו יכול להכיל מפתחות זהים. אם תקרא לשיטה **Add** כדי להוסיף מפתח אשר כבר קיים במערך המפתחות הנוכחי, אתה תקבל חריג, אלא אם כן תשתמש בסוגריים מרובעים כדי להוסיף צמד מפתח-ערך (ראה דוגמה בהמשך). כדי לבדוק אם **Hashtable** מכיל כבר מפתח זה או אחר, תוכל להשתמש בשיטה **ContainsKey**.
- שימוש במשפט **foreach** לשם דפדוף ב-**Hashtable** יגרום להחזרת הערך **DictionaryEntry**. המחלקה **DictionaryEntry** מספקת גישה לרכיבי המפתח והערכים הנמצאים בשני המערכים באמצעות המאפיין **Key** והמאפיין **Value**.

הנה דוגמה המקשרת בין הגילאים של חברי המשפחה לבין שמותיהם, ומדפיסה אותם במסך:

```
using System;
using System.Collections;
...
Hashtable ages = new Hashtable();
...
// fill the SortedList
ages["John"] = 41;
ages["Diana"] = 42;
ages["James"] = 13;
ages["Francesca"] = 11;
```

```
...
// iterate using a foreach statement
// the iterator generates a DictionaryEntry object containing a
key/value pair
foreach (DictionaryEntry element in ages)
{
   string name = (string)element.Key;
   int age = (int)element.Value;
   Console.WriteLine("Name: {0}, Age: {1}", name, age);
}
```

קוד זה יפיק את הפלט הבא:

```
Name: James, Age: 13
Name: John, Age: 41
Name: Francesca, Age: 11
Name: Diana, Age: 42
```

המחלקה SortedList

המחלקה **SortedList** דומה מאוד למחלקה **Hashtable** מהבחינה שהיא מאפשרת לשייך ערכים למפתחות. ההבדל העיקרי הוא שמערך המפתחות תמיד **ממוין** (sorted).

בעת הכנסת צמד מפתח-ערך למערך **SortedList**, המפתח מאוחסן במקום הנכון במערך המפתחות כדי לא לפגוע בסדר המיון שנקבע. לאחר מכן הערך מאוחסן במערך הערכים תחת אותו אינדקס. המחלקה **SortedList** מוודאת באופן אוטומטי שהמפתחות והערכים יאוחסנו בהתאמה זה לזה, גם בעת הוספה והסרה של רכיבים. משמעות הדבר היא שניתן להכניס צמדי מפתח-ערך ל-**SortedList** בכל סדר רצוי, והרשימה תסודר תמיד לפי ערכי המפתחות.

בדומה למחלקה **Hashtable**, גם **SortedList** אינה יכולה להכיל מפתחות זהים. שימוש במשפט **foreach** לדפדוף ב-**SortedList** יגרום להחזרת **DictionaryEntry**. אולם, האובייקטים שנמצאים ב-**DictionaryEntry** יסודרו על פי המאפיין **Key**.

נציג שוב את הדוגמה המקשרת את הגילאים של בני המשפחה עם שמותיהם, ומדפיסה אותם. הפעם נעשה זאת באמצעות **SortedList** ולא באמצעות **Hashtable** (שים לב שהמפתח הוא השם):

```
using System;
using System.Collections;
...
SortedList ages = new SortedList();
...
// fill the SortedList
ages["John"] = 39;
ages["Diana"] = 40;
ages["James"] = 12;
ages["Francesca"] = 10;
...
// iterate using a foreach statement
// the iterator generates a DictionaryEntry object containing a
   key/value pair
```

```
foreach (DictionaryEntry element in ages)
{
   string name = (string)element.Key;
   int age = (int)element.Value;
   Console.WriteLine("Name: {0}, Age: {1}", name, age);
}
```

הפלט של התוכנית מסודר לפי שמות בני המשפחה:

```
Name: Diana, Age: 40
Name: Francesca, Age: 10
Name: James, Age: 12
Name: John, Age: 39
```

השוואה בין מערכים לאוספים

להלן סיכום ההבדלים החשובים בין מערכים לאוספים:

- מערך מכריז על סוג הפריטים שהוא מכיל, אוסף אינו עושה זאת. הסיבה לכך היא שאוספים מאחסנים את הרכיבים שלהם כאובייקטים.
- מופע של מערך הוא בעל גודל קבוע, ואינו יכול לגדול או להצטמק. אוסף יכול להשתנות באופן דינמי במידת הצורך.
- מערך הוא מבנה נתונים לקריאה וכתיבה, ואין אפשרות ליצור מערך לקריאה בלבד. אוסף ניתן להגדיר לקריאה בלבד באמצעות השיטה **ReadOnly**. שיטה זו מחזירה גרסה לקריאה בלבד של האוסף.

שימוש במחלקות אוסף למשחקי קלפים

התרגיל הבא מציג יישום של Microsoft Windows אשר מדמה חלוקה של חפיסה קלפים בין ארבעה שחקנים. הקלפים יכולים להימצא בחפיסה או אצל אחד מארבעת השחקנים (ידיים – hands). החפיסה והידיים מיוצגות על ידי אובייקטים של **ArrayList**. ייתכן שאתה שואל את עצמך מדוע לא לייצג זאת על ידי מערכים, כי הרי תמיד יש 52 קלפים בחפיסה, ו–13 קלפים ביד. זה היה נכון אילולא העובדה שבעת החלוקה של הקלפים לשחקנים, הם כבר אינם נמצאים בחפיסה. אם תייצג את החפיסה באמצעות מערך, תצטרך לנהל רישום של מספר התאים המכילים קלפים (**PlayingCard**). כמו כן, אם אחד השחקנים יחזיר קלף לחפיסה, תצטרך לנהל רישום של מספר התאים ביד, שאינם מכילים יותר קלפים.

עליך ללמוד ולהבין את הקוד, ולאחר מכן לכתוב שתי שיטות: הראשונה לערבוב חפיסת קלפים והשנייה להחזרת הקלפים מהיד לחפיסה.

חלק את הקלפים

1. הפעל את סביבת הפיתוח Microsoft Visual Studio 2005.
2. פתח את הפרויקט **Cards** שבתיקייה My Documents\Microsoft Press\Visual CSharp Step by Step\Chapter 10\Cards.

3. בתפריט Debug בחר Start Without Debugging.

סביבת הפיתוח תבנה ותפעיל את התוכנית. הטופס Windows יציג את הקלפים שבידי ארבעת השחקנים (North, South, East ו-West). בנוסף, יש שני לחצנים: אחד לחלוקת הקלפים והשני להחזרת הקלפים לחפיסה.

4. בטופס Windows לחץ Deal.

52 הקלפים שבחבילה יחולקו לארבעת השחקנים, 13 לכל שחקן.

ניתן לראות שהקלפים טרם עורבבו. בתרגיל הבא תפעיל את השיטה **Shuffle** (ערבוב).

5. לחץ Return To Pack (החזר לחפיסה).

לא קורה דבר מכיוון שטרם נכתבה השיטה שאמורה להחזיר את הקלפים לחפיסה.

6. לחץ Deal.

הפעם הקלפים בכל אחת מהידיים ייעלמו, כי לפני חלוקת הקלפים כל אחת מהידיים מאופסות. מכיוון שאין קלפים בחפיסה (כי השיטה להחזרת הקלפים טרם נכתבה), אין מה לחלק.

7. סגור את הטופס כדי לחזור לסביבת הפיתוח.

עכשיו שאתה יודע איזה חלקים חסרים ביישום, עליך להשלימם.

ערבוב החפיסה

1. פתח את קובץ המקור Pack.cs בחלון Code and Text Editor.

2. עיין בחלקי הקוד השונים.

המחלקה **Pack** מייצגת חפיסת קלפים. היא מכילה שדה פרטי **cards** מסוג **ArrayList**. שים לב שהמחלקה **Pack** מכילה בנאי אשר יוצר את 52 קלפי המשחק ומוסיף אותם ל-**ArrayList** באמצעות השיטה **Accept** אשר הוגדרה על פי אותה מחלקה. שאר השיטות במחלקה מבצעות פעולות רגילות המקובלות בחפיסת קלפים (**Shuffle**, **Deal** וכו').

קלפי המשחק עצמם מיוצגים על ידי המחלקה **PlayingCard** (שנמצאת ב-PlayingCard.cs). לכל מחלקת קלף יש שני שדות עיקרים: **suit** (מסוג רשימה, שיכול להכיל את הערכים Clubs, Diamond, Heart או Spades. אלה הם סוגי הקלפים השונים) ו-**pips** (שמציין את הערך הנומרי של הקלף).

.3 אתר את השיטה **Shuffle** במחלקה **Pack**.

השיטה אינה מיושמת עדיין.

יש מספר דרכים לדמות ערבוב של חפיסת קלפים. הפשוטה ביניהן היא כנראה בחירה של כל אחד מהקלפים לפי הסדר, והחלפתו עם אחד הקלפים האחרים באופן אקראי. NET Framework. כוללת את המחלקה **Random** שתשמש אותך ליצירת ערכים שלמים אקראיים.

.4 הכרז על משתנה מקומי **random** מסוג **Random** ואתחל אותו לאובייקט אקראי חדש באמצעות בנאי ברירת המחדל **Random** (שים לב ואל תתבלבל, מי זה מי למרות שלכולם יש אותם שמות).

השיטה **Shuffle** נראית כעת כך:

```
public void Shuffle()
{
   Random random = new Random();
}
```

.5 הוסף משפט **for** ללא תוכן, אשר מגדיל באחד את המשתנה **i** מסוג **int** מהערך 0 ועד מספר הרכיבים שבאוסף **ArrayList**.

השיטה **Shuffle** נראית כעת כך:

```
public void Shuffle()
{
   Random random = new Random();
   for (int i = 0; i != cards.Count; i++)
   {
   }
}
```

השלב הבא יהיה בחירת אינדקס אקראי בין 0 ל-**cards.Count**. תצטרך להחליף בין הקלף אשר באינדקס **i** לבין הקלף שבאינדקס האקראי שקיבלת. תוכל לייצר ערך שלם חיובי אקראי באמצעות קריאה לשיטה **Next** ששייכת למופע **random**. מרבית הסיכויים שערך אותו שלם חיובי יהיה גדול מ-**cards.Count**. לכן עליך להמיר תחילה ערך זה לערך שבין 0 לבין "**cards.Count** – 1" בלבד. הדרך הפשוטה ביותר לעשות זאת היא להשתמש באופרטור % כדי למצוא את השארית המתקבלת כאשר מחלקים את המספר ב-**cards.Count**.

.6 בתוך הלולאה **for**, הכרז על משתנה מקומי **cardToSwap** ואתחל אותו למספר אקראי בין 0 ל- **cards.Count**.

כעת, השיטה **Shuffle** נראית בדיוק כך:

```
public void Shuffle()
{
   Random random = new Random();
   for (int i = 0; i != cards.Count; i++)
   {
      int cardToSwap = random.Next() % cards.Count;
   }
}
```

השלב האחרון הוא החלפת הקלף המאוחסן באינדקס **i** עם הקלף המאוחסן באינדקס **cardToSwap**. כדי לעשות זאת, עליך להשתמש במשתנה מקומי זמני. זכור שהרכיבים בתוך מחלקת אוסף (כגון **ArrayList**) הם **אובייקטים** מסוג **System.Object**.

7. הוסף שלושה משפטים אשר יחליפו את הקלף באינדקס **i** עם הקלף באינדקס **cardToSwap**.

כעת, השיטה **Shuffle** נראית בדיוק כך:

```
public void Shuffle()
{
   Random random = new Random();
   for (int i = 0; i != cards.Count; i++)
   {
      int cardToSwap = random.Next() % cards.Count;
      object temp = cards[i];
      cards[i] = cards[cardToSwap];
      cards[cardToSwap] = temp;
   }
}
```

שים לב שעליך להשתמש במשפט **for**. משפט **foreach** לא יפעל כראוי, כי עליך לשנות כל אחד מהרכיבים שב-**ArrayList**, והמשפט **foreach** מאפשר קריאה בלבד.

8. בתפריט Debug בחר Start Without Debugging.

סביבת הפיתוח תבנה ותפעיל את התוכנית.

9. לחץ Deal.

החפיסה מעורבבת כעת ומוכנה לחלוקה (הקלפים יחולקו בכל פעם באופן שונה, כי סדר הקלפים אקראי).

10. סגור את הטופס.

השלב האחרון בפעילות הוא הוספת הקוד אשר יחזיר את הקלפים לחפיסה.

החזרת הקלפים לחפיסה

1. פתח את קובץ המקור Hand.cs בחלון Code and Text Editor.

המחלקה **Hand**, אשר מכילה גם את האוסף **ArrayList** הנקרא **cards**, מייצגת את הקלפים שבידי השחקן. הרעיון הוא שבכל זמן נתון כל קלף נמצא בחפיסה או אצל אחד השחקנים.

2. אתר את השיטה **ReturnCardsTo** במחלקה **Hand**.

המחלקה **Pack** כוללת את השיטה **Accept** שמקבלת פרמטר יחיד מסוג **PlayingCard**. עליך ליצור לולאה אשר תסקור את הקלפים שביד ותחזיר אותם לחפיסה.

3. הוסף פקודות לשיטה **ReturnCardsTo** וערוך אותה באופן הבא:

```
public void ReturnCardsTo(Pack pack)
{
   foreach (PlayingCard card in cards)
   {
      pack.Accept(card);
   }
   cards.Clear();
}
```

משפט **foreach** מתאים לצרכינו מכיוון שאינך צריך לערוך את הרכיבים ואין צורך לדעת את האינדקס שלהם. השיטה **Clear** מוחקת את כל הרכיבים באוסף. חשוב לקרוא לשיטה **cards.Clear** לאחר החזרת הקלפים לחפיסה, כדי שלא יהיו קלפים כפולים ביד ובחפיסה.

4. בתפריט Debug בחר Start Without Debugging.

5. לחץ Deal.

הקלפים המעורבבים יחולקו לארבעת השחקנים כמקודם.

6. לחץ Return To Pack.

הידיים יתרוקנו מקלפים. כעת כל הקלפים שוב בחבילה.

7. לחץ Deal.

הקלפים המעורבבים יחולקו שוב לארבעת השחקנים.

8. סגור את הטופס.

הערה

אם תלחץ פעמיים על Deal מבלי ללחוץ על Return To Pack (החזרה לחפיסה), תאבד את כל הקלפים. ביישום אמיתי תצטרך לחסום את הלחצן Deal עד שתזהה לחיצה על Return To Pack. בחלק ד' של הספר נלמד עוד על יצירת קוד של ממשק-משתמש.

בפרק זה למדת ליצור מערכים ולהשתמש בהם כדי לערוך קבוצות של נתונים. למדת להשתמש בחלק ממחלקות האוסף כדי לאחסן מידע בזיכרון וגם לגשת אליו במספר דרכים שונות.

- **אם ברצונך להמשיך לפרק הבא**

  השאר את Visual Studio 2005 פעילה ועבור לפרק 11.

- **אם ברצונך לכבות כעת את Visual Studio 2005**

  פתח את תפריט Menu ולחץ Exit. אם מופיעה תיבת דו-שיח Save, לחץ Yes.

פרק 10 – טבלה מסכמת

המשימה	צריך
להכריז על משתנה מערך.	לכתוב את שם סוג הרכיב (element), לאחריו סוגריים מרובעים, לאחריהם שם המשתנה ולבסוף נקודה-פסיק. לדוגמה: `bool[] flags;`
ליצור מופע של מערך.	לכתוב את מילת המפתח **new**, אחריה שם סוג הרכיב ואחריו את גודל המערך בתוך סוגריים מרובעים. לדוגמה: `bool[] flags = new bool[10];`
לאתחל את רכיבי המופע של המערך לערכים מסוימים.	לכתוב את הערכים בין סוגריים מסולסלים ולהפריד ביניהם באמצעות פסיקים. לדוגמה: `bool[] flags = { true, false, true, false };`
למצוא את מספר הרכיבים במערך.	להשתמש במאפיין **Length**. לדוגמה: `int noOfElements = flags.Length;`
לגשת לאחד מרכיבי המערך.	לכתוב את שם משתנה המערך ואחריו מספר האינדקס של הרכיב בין סוגריים מרובעים. זכור שאינדקס של מערך מתחיל באפס ולא באחד. לדוגמה: `bool initialElement = flags[0];`

לדפדף ברכיבי מערך או אוסף.	להשתמש במשפט **for** או **foreach**. לדוגמה:
	`bool[] flags = { true, false, true, false };`
	`for (int i = 0; i != flags.Length; i++)`
	`{`
	`  Console.WriteLine(flags[i ]);`
	`}`
	
	`foreach (bool flag in flags)`
	`{`
	`  Console.WriteLine(flag);`
	`}`

למצוא את מספר הרכיבים באוסף.	להשתמש במאפיין **Count**. לדוגמה: `int noOfElements = flags.Count;`

פרק 11

מערכי פרמטר

בסיום פרק זה, תוכל:

- לכתוב שיטה שיכולה לקבל מספר בלתי מוגבל של ארגומנטים באמצעות מילת המפתח **params**.
- לכתוב שיטה אשר יכולה לקבל מספר בלתי מוגבל של פרמטרים מכל סוג באמצעות מילת המפתח **params** בשילוב עם סוג אובייקט.

מערכי פרמטר (parameter arrays) שימושיים ליצירת שיטות שיכולות לקבל מספר לא קבוע של ארגומנטים מסוג אחד או יותר, בתור פרמטרים. אם עסקת בעבר בתכנות מונחה עצמים, המשפט האחרון עשוי להיראות לך חשוד במקצת. אחרי הכל, הפתרון שמציע תכנות מונחה עצמים לבעיה זו הוא **שיטות מועמסות** (overloaded methods).

העמסה (**overloading**) הוא מושג טכני שמשמעותו הכרזה על שתי שיטות או יותר בעלות אותו שם ובאותו תחום הכרזה (scope). היכולת להעמיס שיטה שימושית מאוד במקרים שבהם ברצונך לבצע את אותה הפעולה על ארגומנטים מסוגים שונים. הדוגמה הקלאסית להעמסה ב- Microsoft Visual C# היא השיטה **Console.WriteLine**. השיטה **WriteLine** מועמסת מספר כה רב של פעמים, שניתן להעביר לה כארגומנט כל סוג ערך פרימיטיבי:

```
class Console
{
   public static void WriteLine(int parameter)
   ...
   public static void WriteLine(double parameter)
   ...
   public static void WriteLine(decimal parameter)
   ...
}
```

למרות שעיקרון העמסה שימושי ביותר, הוא אינו יכול לפתור את כל המקרים. קשה להעמסה להתמודד עם מקרים שבהם סוג הפרמטר אינו משתנה, אולם מספר הפרמטרים כן. לדוגמה, מה ניתן לעשות במצב שבו אתה רוצה להדפיס ערכים רבים על המסך. האם עליך ליצור גרסה של **Console.WriteLine** אשר מקבלת שני פרמטרים, גרסה נוספת המקבלת שלושה פרמטרים, וכד'? העתקה מרובה זו של שיטות מועמסות אינה פתרון טוב. למרבה המזל, קיימת דרך לכתוב שיטה אחת אשר יכולה לקבל מספר משתנה של ארגומנטים (**variadic method**) באמצעות מערך פרמטר (פרמטר שהוכרז בעזרת מילת המפתח **params**).

כדי להבין כיצד מערכי **params** מסייעים לפתרון הבעיה, חשוב להכיר תחילה את השימוש במערכים רגילים ואת החסרונות שלהם.

ארגומנטים מסוג מערך

נניח שברצונך לכתוב שיטה אשר תמצא את הערך הנמוך ביותר מבין קבוצה של פרמטרים. דרך אחת לעשות זאת תהיה להשתמש במערך. לדוגמה, כדי למצוא את הערך הנמוך ביותר מבין מספר ערכים שלמים, ניתן לכתוב שיטה סטטית בשם **Min** המקבלת פרמטר אחד המייצג מערך של ערכי **int**:

```
class Util
{
   public static int Min(int[] paramList)
   {
      if (paramList == null || paramList.Length == 0)
      {
         throw new ArgumentException("Util.Min");
      }
      int currentMin = paramList [0];
      foreach (int i in paramList)
      {
         if (i < currentMin)
         {
            currentMin = i;
         }
      }
      return currentMin;
   }
}
```

כדי למצוא את הערך הקטן ביותר מבין שני ערכי **int** באמצעות השיטה **Min** תכתוב:

```
int[] array = new int[2];
array[0] = first;
array[1] = second;
int min = Util.Min(array);
```

כדי למצוא את הערך הקטן ביותר מבין שלושה ערכי int באמצעות השיטה min תכתוב:

```
int[] array = new int[3];
array[0] = first;
array[1] = second;
array[2] = third;
int min = Util.Min(array);
```

פתרון זה מבטל את הצורך במספר רב של העמסות (overload), אולם לפתרון זה יש מחיר: עליך לכתוב את הקוד אשר מציב ערכים במרכיבים השונים במערך המוזן לשיטה. עם זאת, ניתן להעביר חלק מהעומס של כתיבת קוד זה למהדר באמצעות שימוש במילת המפתח **params** להכרזה על מערך **params**.

הכרזה על מערכי params

מילת המפתח **params** משמשת לעריכת פרמטרים מסוג מערך. כדי להדגים, נשתמש שוב בשיטה **Min**, אך הפעם, פרמטר המערך שלה יוגדר כפרמטר **params**:

```
class Util
{
   public static int Min(params int[] paramList)
   {
      // code exactly as before
   }
}
```

השפעת מילת המפתח **params** על השיטה **Min** היא האפשרות לקרוא לשיטה באמצעות כל מספר של ארגומנטים מסוג **int**.

```
int min = Util.Min(first, second);
```

המהדר יתרגם את הקריאה לקוד אשר נראה דומה לדוגמה זו:

```
int[] array = new int[2];
array[0] = first;
array[1] = second;
int min = Util.Min(array);
```

כדי למצוא את הנמוך מבין שלושה ערכים שלמים, עליך לכתוב את הקוד שלהלן. גם הוא מומר על ידי המהדר לקוד מתאים אשר משתמש במערך:

```
int min = Util.Min(first, second, third);
```

שתי הקריאות למחלקה **Min** (קריאה אחת עם שני ארגומנטים ואחת עם שלושה ארגומנטים) מפעילות את אותה שיטה **Min** באמצעות מילת המפתח **params**. כלומר, ניתן לקרוא לשיטה **Min** עם מספר בלתי מוגבל של ארגומנטים. המהדר סופר את מספר הארגומנטים, יוצר מערך של ערכי **int** באותו הגודל, מציב את הארגומנטים במערך ולבסוף קורא לשיטה ומציב בה פרמטר אחד מסוג מערך.

הערה

מתכנתים מרקע תכנות של C ו-++C עשויים לראות במילת המפתח **params** את המקבילה למאקרו **varargs** מקובץ הכותרת stdarg.h.

- ניתן להשתמש במילת המפתח **params** רק עם מערכים חד-ממדיים, כפי שמוצג בדוגמה הבאה:

```
// compile-time error
public static int Min(params int[,] table)
...
```

- אין להשתמש במערכי פרמטר מסוג **ref** או **out** כפי שמוצג בדוגמה הבאה:

```
// compile-time errors
public static int Min(ref params int[] paramList)
...
public static int Min(out params int[] paramList)
...
```

- מערך **params** חייב להיות הפרמטר האחרון של השיטה (כלומר, שיטה אינה יכולה לקבל יותר ממערך **params** אחד). בחן את הדוגמה הבאה:

```
// compile-time error
public static int Min(params int[] paramList, int i)
...
```

- לשיטה שאינה **params** תמיד תהיה עדיפות על שיטת **params**. כלומר, ניתן על פי הצורך ליצור גרסה מועמסת של שיטה עבור המקרים השכיחים. לדוגמה:

```
public static int Min(int leftHandSide, int rightHandSide)
...
public static int Min(params int[] paramList)
...
```

הגרסה הראשונה של השיטה **Min** מופעלת כאשר בקריאה לשיטה הופיעו שני ארגומנטים מסוג **int**. הגרסה השנייה מופעלת עבור כל מספר אחר של ארגומנטים מסוג **int**, או במקרה שהקריאה לשיטה אינה מכילה ארגומנטים כלל.

הוספת גרסאות מועמסות שאינן **params** לשיטה, עשויה לשפר את הביצועים, כי המהדר לא יצטרך ליצור כל כך הרבה מערכים ולהציב בהם ערכים.

שימוש ב- params object[ ]

מערך **params int** שימושי ביותר, כי הוא מאפשר להעביר מספר בלתי מוגבל של ארגומנטים לשיטה. אולם, מה קורה כאשר לא רק כמות הארגומנטים משתנה, אלא גם סוגם? שפת C# מספקת פתרון גם לבעיה זו. הטכניקה מבוססת על כך ש-**System.Object** היא מחלקת השורש (**object**) של כל המחלקות, ושהמהדר יכול ליצור קוד אשר ממיר סוגי ערך (שאינם מחלקות) לאובייקטים באמצעות אריזה (boxing), כפי שמוסבר בפרק 8. ניתן להשתמש במערכי object **params** כדי להכריז על שיטות שיכולות לקבל מספר בלתי מוגבל של ארגומנטים מסוג **object**, וכך לאפשר לארגומנטים המועברים למחלקה להיות מכל סוג שהוא. עיין בדוגמה הבאה:

```
class Black
{
   public static void Hole(params object [] paramList)
   ...
}
```

השיטה נקראת **Black.Hole**, מכיוון שאף ארגומנט אינו יכול לחמוק ממנה, ולא מכיוון שהיא יכולה לטפל בכל סוג של ארגומנט:

- כאשר לא מעבירים ארגומנטים כלל, המהדר יעביר מערך אובייקטים שאורכו 0:

```
Black.Hole();
// converted into Black.Hole(new object[0]);
```

טיפ

אין קושי לדפדף במערך שאורכו 0 באמצעו משפט **foreach**.

- ניתן לקרוא לשיטה באמצעות העברת ארגומנט **null**. מערך הוא מסוג הפניה, ולכן ניתן לאתחל אותו לערך **null**:

```
Black.Hole(null);
```

- ניתן להעביר מערך ממשי. במילים אחרות, ניתן ליצור באופן עצמאי את המערך אשר המהדר אמור ליצור:

```
object[] array = new object[2];
array[0] = “forty two”;
array[1] = 42;
Black.Hole(array);
```

- ניתן להעביר למערך כל ארגומנט אחר מכל סוג שהוא. ארגומנטים אלו יאוחסנו באופן אוטומטי במערך **object**:

```
Black.Hole(“forty two”, 42);
//converted into Black.Hole(new object[]{“forty two”, 42});
```

השיטה Console.WriteLine

המחלקה **Console** מכילה מספר רב של העמסות של השיטה **WriteLine**. אחת מהן נראית כך:

```
public static void WriteLine(string format, params object[] arg);
```

הנה דוגמה לקריאה לשיטה הזו, אשר הופיעה כבר בפרק 10:

```
Console.WriteLine("Name:{0}, Age:{1}", name, age);
```

המהדר מפרש את הקריאה באופן הבא:

```
Console.WriteLine("Name:{0}, Age:{1}", new object[2]{name, age});
```

מערכי params

בתרגיל הבא עליך לכתוב שיטה סטטית בשם **Util.Sum** ולבדוק אותה. מטרת השיטה לחשב את ערך הסכום של מספר לא קבוע של ארגומנטים מסוג **int** המועברים לה, ולהחזיר את הסכום כערך int. כדי לעשות זאת, השיטה **Util.Sum** מקבלת פרמטר **params int[]**. עליך להוסיף שתי בדיקות עבור הפרמטר **params** אשר יבטיחו את יציבות השיטה **Util.Sum**. לבסוף, השיטה **Util.Sum** תקרא באמצעות מגוון של ארגומנטים שונים כדי לבדוק את פעולתה.

כתוב שיטה עבור מערך params

1. הפעל את סביבת הפיתוח Visual Studio 2005.
2. פתח את הפרויקט ParamsArrays שבתיקייה My Documents\Microsoft Press\Visual CSharp Step by Step\Chapter 11\ ParamArrays.
3. הצג את קובץ המקור Util.cs בחלון Code and Text Editor.
   קובץ זה מכיל מחלקה ריקה בשם **Util** ששייכת למרחב השמות **ParamsArray**.
4. הוסף למחלקה **Util** שיטה סטטית ציבורית בשם **Sum**. השיטה **Util.Sum** מחזירה ערך **int** ומקבלת מערך **params** של ערכי **int**. השיטה נראית כך:

```
class Util
{
   public static int Sum(params int[] paramList)
   {
   }
}
```

השלב הראשון בבניית השיטה **Util.Sum** הוא בדיקת הפרמטר **paramList**. אם הוא אינו מכיל מערך חוקי של ערכי **int**, הוא עשוי להיות **null** או מערך שאורכו אפס. בשני המקרים הללו קשה לחשב את הסכום, והפתרון האידיאלי יהיה לזרוק חריג

ArgumentException (ניתן לטעון שסכום הערכים במערך שאורכו אפס הוא אפס, אולם בדוגמה זו נתייחס למצב זה כחריג).

הערה

המחלקה **ArgumentException** מיועדת להיזרק מתוך מחלקה כאשר הארגומנט שהועבר למחלקה אינו תואם לדרישות שלה.

5. הוסף משפט לשיטה **Util.Sum** אשר זורק **ArgumentException** אם **paramList** הוא **null**. כעת השיטה נראית כך:

```
public static int Sum(params int[] paramList)
{
   if (paramList == null)
   {
      throw new ArgumentException("Util.Sum: null parameter
      list");
   }
}
```

6. הוסף משפט לשיטה **Util.Sum** אשר זורק **ArgumentException** אם אורכו של המערך הוא 0. כעת השיטה נראית כך:

```
public static int Sum(params int[] paramList)
{
   if (paramList == null)
   {
      throw new ArgumentException("Util.Sum: null parameter
      list");
   }
   if (paramList.Length == 0)
   {
      throw new ArgumentException("Util.Sum: empty parameter
      list");
   }
}
```

כאשר המערך עובר את שתי הבדיקות הללו, השלב הבא יהיה חיבור של כל מרכיביו. ניתן להיעזר במשפט **foreach** לביצוע פעולה זו. עליך גם להכריז על משתנה מקומי לאחסון ערך הסכום.

7. הוסף משפט **foreach** לשיטה **Util.Sum** כדי לחבר את כל מרכיבי המערך, והחזר את הערך באמצעות משפט החזרה (**return**).

כעת השיטה נראית כך:

```
class Util
{
   public static int Sum(params int[] paramList)
   {
      ...
      int sumTotal = 0;
```

```
        foreach (int i in paramList)
        {
           sumTotal += i;
        }
        return sumTotal;
    }
}
```

8. בתפריט Build בחר Build Solutions. ודא שהקוד אינו מכיל שגיאות.

בדיקת השיטה Util.Sum

1. בחלון Code and Text Editor פתח את קובץ המקור Program.cs.
2. בחלון Code and Text Editor, אתר את השיטה **Entrance** במחלקה **Class**.
3. הוסף את המשפט הבא לשיטה **Entrance**:

```
Console.WriteLine(Util.Sum(null));
```

4. בתפריט Debug בחר Start Without Debugging.
התוכנית תיבנה ותופעל, ואז תוצג במסך ההודעה הבאה:

```
Exception: Util.Min: null parameter list
```

מהודעה זו משתמע שהבדיקה הראשונה בשיטה פועלת.

5. הקש Enter כדי לסגור את התוכנית ולחזור ל- Visual Studio 2005.
6. בחלון Code and Text Editor שנה את משפט הקריאה לשיטה **Console.WriteLine** שבשיטה **Entrance**, לפי דוגמה זו:

```
Console.WriteLine(Util.Sum());
```

הפעם, הקריאה לשיטה מתבצעת ללא כל ארגומנט. המהדר יבחין שאין ארגומנטים, וייצור מערך ריק.

7. בתפריט Debug בחר Start Without Debugging.
התוכנית תיבנה ותופעל, וכתוצאה תוצג במסך ההודעה הבאה:

```
Exception: Util.Min: empty parameter list
```

מהודעה זו משתמע שהבדיקה השנייה שבשיטה גם פועלת.

8. הקש Enter כדי לסגור את התוכנית ולחזור לסביבת הפיתוח.
9. שנה את משפט הקריאה לשיטה **Console.WriteLine** שבשיטה **Entrance**, לפי דוגמה זו:

```
Console.WriteLine(Util.Sum(10,9,8,7,6,5,4,3,2,1));
```

10. בתפריט Debug בחר Start Without Debugging.
התוכנית תיבנה, תופעל, ותדפיס 55 על המסך.

11. הקש Enter כדי לסגור את היישום.

בפרק זה למדת להשתמש במערך **params** כדי להגדיר שיטה שיכולה לקבל מספר לא קבוע של ארגומנטים. למדת גם להשתמש במערך **params** של סוגי **object** כדי להעביר לשיטה ארגומנטים מסוגים שונים.

☯ **אם ברצונך להמשיך לפרק הבא**

השאר את Visual Studio 2005 פועלת ועבור לפרק 12.

☯ **אם ברצונך לכבות כעת את Visual Studio 2005**

פתח את תפריט Menu ולחץ Exit. כאשר תוצג תיבת הדו-שיח Save, לחץ Yes.

פרק 11 – טבלה מסכמת

המשימה	צריך

לכתוב שיטה המקבלת מספר בלתי מוגבל של ארגומנטים מסוג מסוים.	לכתוב שיטה המקבלת כפרמטר מערך **params** מסוג מסוים. לדוגמה, שיטה המקבלת מספר בלתי מוגבל של ארגומנטים מסוג **bool** תיראה כך:
	`someType Method(params bool[] flags)`
	`{`
	`  ...`
	`}`

לכתוב שיטה המקבלת מספר בלתי מוגבל של ארגומנטים מכל סוג.	לכתוב שיטה המקבלת כפרמטר מערך **params** מסוג **object**. לדוגמה:
	`someType Method(params object[] paramList)`
	`{`
	`  ...`
	`}`

פרק 12

עבודה עם הורָשָה (inheritance)

בסיום פרק זה, תוכל:

- ליצור מחלקה נגזרת (derived class) אשר יורשת מאפיינים ממחלקת בסיס (base class).
- לשלוט בהסתרה (hiding) ובעקיפה (overriding) של שיטות באמצעות מילות המפתח **new**, **virtual** ו-**override**.
- להגביל את הגישה במסגרת של היררכיית הורשה (inheritance hierarchy) באמצעות מילת המפתח **protected**.
- לאגד אלמנטים משותפים של מימושים שונים במחלקה מופשטת (abstract class).
- להכריז שלא ניתן לרשת ממחלקה מסוימת באמצעות מילת המפתח **sealed**.
- ליצור ממשק אשר מזהה את שמות השיטות.
- לממש ממשקים במבנים או במחלקות באמצעות כתיבת גוף השיטות.

הורשה (**inheritance**) הינה רעיון מרכזי בעולם המונחה עצמים. ניתן להשתמש בה בכדי להימנע מחזרות בעת הגדרת מחלקות בעלות מאפיינים משותפים, ואשר קשורות זו לזו באופן ברור. ייתכן שיהיו אלו מחלקות שונות מאותו הסוג, אשר לכל אחת מהן מאפיין ייחודי משלה. למשל, **מנהלים**, **פועלים**, או **כל העובדים** במפעל. אם תכתוב ישות המייצגת מפעל, כיצד תציין שלמנהלים ולעובדים יש מספר מאפיינים משותפים, וגם מספר מאפיינים אשר שונים זה מזה? לדוגמה, לכולם יש מספר-עובד, אולם המנהלים צריכים לבצע פעולות ומשימות שונות מאלו של הפועלים. במקרים כאלה מנגנון ההורשה שימושי ויעיל ביותר.

מהי הורשה?

קיים בלבול רב בקרב המתכנתים לגבי ההגדרה המדויקת למונח הורשה. בעייתיות זו נובעת מהעובדה שלמילה הורשה (inheritance) עצמה יש מספר משמעויות שההבדל ביניהן דק ביותר. אם מישהו משאיר לך משהו בצוואתו, הוא מוריש לך ואתה יורש אותו. כמו כן, כל אדם יורש חצי מהגנים שלו מאמו וחצי מהגנים מאביו. לשתי הדוגמאות והמשמעויות הללו אין כל דבר משותף עם המושג **הורשה** שבתכנות.

הורשה בתכנות עוסקת בעיקר בסיווג, ועל כן, הורשה מייצגת את היחס בין שתי מחלקות. לדוגמה, בבית הספר למדת על יונקים, וגם על כך שסוסים ולוויתנים הם סוגים של יונקים.

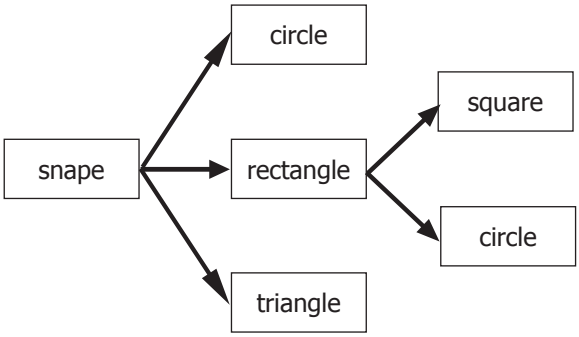
כל אחד מהם מבצע את אותן הפעולות המשותפות לכל היונקים (נשימת אוויר, המלטת ולדות חיים, ועוד), עם זאת, לכל אחד מהם יש גם מאפיינים מיוחדים משלו (לדוגמה, לסוס יש פרסות, בניגוד ללוויתן).

בשפת #C ניתן לדמות את היחס הזה על ידי יצירת שתי מחלקות, אחת בשם **Mammal** והשנייה בשם **Horse**, ולהכריז שהמחלקה **Horse** יורשת מהמחלקה **Mammal** (תכונות, כמובן). ההורשה תדמה את היחס הקיים בין השנים ואת העובדה **שכל** הסוסים הם למעשה יונקים. תוכל להכריז על מחלקה נוספת בשם **Whale** היורשת גם היא מהמחלקה **Mammal**. ניתן יהיה להגדיר מכנים משותפים (כגון נשימה והמלטה) במחלקה **Mammal**, בעוד שמאפיינים כגון פרסות וסנפירים יוגדרו במחלקות **Horse** ו-**Whale** בהתאמה.

כל אחד מאתנו יורש תכונות מהוריו, כגון צבע עיניים, צבע שיער ועוד. בנוסף לתכונות אלו, לכל אחד יש תכונות משלו בנוסף לתכונות שעוברות בתורשה מן ההורים. כפי שאנו גוזרים תכונות מהורינו ומרחיבים אותם, גם מחלקות ניתנות לגזירה ממחלקות אחרות. חלק מתכונות אלו נדרסות על ידי תכונות משלנו וחלקן לא.

הורשה מאפשרת ליצור מחלקה חדשה, המבוססת על מחלקה קיימת. המחלקה החדשה יכולה להשתמש בכל התכונות של המחלקה המקורית, היא יכולה לדרוס תכונות קיימות, וגם יכולה להרחיב תכונות קיימות או להוסיף תכונות משלה.

מספר מחלקות חדשות יכולות לרשת ממחלקת מקור אחת. עם זאת, ניתן לרשת ממחלקה אחת בלבד. לדוגמה, נתונה מחלקת בסיס בשם **control**, אשר ניתן להוריש ממנה למספר סוגים שונים של פקדים. שים לב שבקריאת התרשים משמאל לימין יכול להיות יחס של אחד לרבים. עם זאת, בקריאה מימין לשמאל יש מחלקת מקור אחת בלבד.



יש מספר מונחים בסיסיים שכיחים בהקשר של הורשה בתכנות:

מחלקת בסיס (base class) – המחלקה המקורית או מחלקת בסיס.

מחלקת הורה (אב, parent class) – שם אחר למחלקת בסיס.

מחלקה נגזרת (derived class) – מחלקה חדשה, נוצרה על ידי הורשה ממחלקת בסיס.

מחלקת בן (child class) – שם אחר למחלקה נגזרת.

הורשה יחידה (single inheritance) – מחלקה נגזרת נוצרת ממחלקת בסיס אחת, ורק אחת. שפת #C תומכת בהורשה יחידה בלבד. התרשים שלעיל ממחיש הורשה יחידה.

הורשה מרובה (multiple inheritance) – מחלקה נגזרת נוצרת משתי מחלקות בסיס, או יותר. שפת #C אינה תומכת בהורשה מרובה.

שימוש בהורשה

בסעיף הבא נציג את התחביר החיוני שדרוש ליצירת מחלקות היורשות ממחלקות אחרות.

מחלקות בסיס ומחלקות נגזרות

תחביר אשר משמש להכרזה על מחלקה אשר יורשת ממחלקה אחרת:

```
class DerivedClass : BaseClass {
...
}
```

המחלקה הנגזרת (derived class) יורשת ממחלקת הבסיס (base class). בניגוד לשפות אחרות, כגון C ו-++C, מחלקה מסוימת יכולה לרשת ממחלקה אחת לכל היותר **ואסור** למחלקה לרשת משתי מחלקות או יותר. עם זאת, כאשר **DerivedClass** מוכרזת כמחלקה **חתומה** (**sealed**) (ראה "מחלקות חתומות" בהמשך פרק זה), ניתן ליצור מחלקות נגזרות אשר יורשות מ-**DerivedClass** באמצעות תחביר דומה:

```
class DerivedSubClass : DerivedClass {
...
}
```

באופן זה, ניתן להכריז על היררכיות הורשה.

נניח שעליך לכתוב מנתח תחבירי (syntax analyzer) כחלק מהמהדר. תצטרך להגדיר מחלקה עבור כל אחד ממרכיבי התוכנית בהתאם לכללים ולשפת התחביר. מרכיבים אלה נקראים לעיתים קרובות **יחידות לקסיקליות** (**tokens**). באפשרותך להגדיר מחלקה **Token** באופן המוצג להלן. הבנאי יבנה אובייקט **Token** באמצעות המחרוזת שהועברה לו, ואשר מייצגת מילת מפתח, מזהה, מקטע של תווים לבנים, או כל מרכיב תחבירי אחר בשפת התכנות הרלוונטית:

```
class Token
{
   public Token(string name)
   {
      ...
   }
   ...
}
```

אחר כך תוכל להגדיר מחלקות עבור כל סוג של יחידה לקסיקלית, בהתבסס על המחלקה **Token**, ותוך כדי הוספה של שיטות במידת הצורך. לדוגמה:

```
class IdentifierToken : Token
{
   ...
}
```

טיפ

מתכנתים מרקע של ++C צריכים לשים לב שלא ניתן לציין אם ההורשה היא ציבורית, פרטית או מוגנת. הורשה בשפת #C לעולם תהיה ציבורית באופן מפורש. מתכנתים מרקע של Java צריכים לשים לב לשימוש בנקודתיים, ולכן שאין שימוש במילת המפתח **extends**.

זכור שהמחלקה **System.Object** הינה מחלקת השורש של כל המחלקות, ולכן כל המחלקות נגזרות ממנה. לדוגמה, נניח שכתבת את המחלקה **Tokens** באופן הבא:

```
class Token
{
   public Token(string name)
   {
      ...
   }
   ...
}
```

במקרה זה, המהדר לא יודיע לך דבר, אבל ישכתב את הקוד באופן הבא (תוכל גם לכתוב אותו בעצמך אם ממש תרצה):

```
class Token : System.Object
{
   public Token(string name)
   {
      ...
   }
   ...
}
```

המשמעות המעשית היא שכל המחלקות שתגדיר יורשות באופן אוטומטי את כל המאפיינים של המחלקה **System.Object**. הדבר תקף גם לגבי שיטות כגון **ToString** (ראה פרק 2), שתפקידה להמיר אובייקטים למחרוזות.

קריאה לבנאים של מחלקות בסיס

כל המחלקות כוללות בנאי אחד לפחות. ואם לא תכתוב בנאי בעצמך, המהדר יצור בעצמו בנאי ברירת מחדל. מחלקה נגזרת כוללת באופן אוטומטי את כל השדות של מחלקת הבסיס. את השדות הללו יש לאתחל בעת יצירת אובייקט. לפיכך, הבנאי של המחלקה הנגזרת חייב לקרוא לבנאי של מחלקת הבסיס. בעת הגדרת בנאי עליך להשתמש במילת המפתח base בכדי לקרוא לבנאי של מחלקת הבסיס. הנה דוגמה:

```
class IdentifierToken : Token
{
   public IdentifierToken(string name)
         : base(name) // calls Token(name)
   {
      ...
   }
   ...
}
```

אם לא תקרא לבנאי של מחלקת הבסיס במסגרת הבנאי של המחלקה הנגזרת, המהדר ינסה להוסיף בעצמו קריאה לבנאי ברירת המחדל של מחלקת הבסיס. אם נשתמש למשל, בדוגמה הקודמת, המהדר היה משכתב את זה:

```
class Token
{
   public Token(string name)
   {
      ...
   }
   ...
}
```

כתוצאה היה מתקבל:

```
class Token : System.Object
{
   public Token(string name) : base()
   {
      ...
   }
   ...
}
```

הסיבה לכך שהקוד יעבוד כהלכה, היא שהמחלקה **System.Class** מכילה בנאי ברירת מחדל **ציבורי**. אולם לא לכל המחלקות יש בנאי ברירת מחדל ציבורי, ובמקרים כאלה השמטת הקריאה לבנאי מחלקת הבסיס תגרום לשגיאה בשלב ההידור, מכיוון שהמהדר חייב להוסיף קריאה לבנאי, אולם אינו יודע באיזה בנאי להשתמש. לדוגמה:

```
class IdentifierToken : Token
{
   public IdentifierToken(string name)
   // error, base class Token does not have
   // a public default constructor
   {
      ...
   }
   ...
}
```

הצבה במחלקות

בדוגמאות קודמות בספר זה למדת להכריז על משתנים מסוג מחלקה, ולהשתמש במילת המפתח **new** כדי ליצור אובייקט. למדת גם כיצד הכללים לבדיקת סוג של C# מונעים ממך להציב משתנים מסוג מסוים באובייקטים אשר נוצרו מתוך סוג אחר. לדוגמה, לאור ההגדרות של המחלקות **Token**, **IdentifierToken** ו-**KeywordToken**, הקוד הבא אינו חוקי:

```
class Token
{
...
}
```

```
class IdentifierToken : Token
{
   ...
}

class KeywordToken : Token
{
   ...
}

...
IdentifierToken it = new IdentifierToken();
KeywordToken kt = it; // error - different types
```

עם זאת, משתנה יכול להפנות לאובייקט מסוג אחר כל עוד הסוג שלו גבוה יותר בהיררכיית ההורשה. אי לכך, המשפט הבא הינו חוקי:

```
IdentierToken it = new IdentifierToken();
Token t = it; // legal as Token is a base class of IdentifierToken
```

משמעות היררכיית ההורשה היא, שניתן לראות ב-**IdentifierToken** סוג מיוחד של Token (הוא כולל את כל המאפיינים של **Token**), הכולל מספר מאפיינים נוספים אשר מוגדרים על ידי השיטות והשדות שנוספו לו. ניתן גם לגרום למשתנה **Token** להפנות לאובייקט **KeywordToken**, אך יש הגבלה חשובה אחת: כאשר מפנים לאובייקט **KeywordToken** או **IdentifierToken** באמצעות משתנה מסוג **Token**, ניתן לגשת אך ורק לשיטות ולשדות אשר הוגדרו במסגרת המחלקה **Token**. שיטות נוספות אשר הוגדרו על ידי המחלקות **KeywordToken** או **IdentifierToken** תהיינה זמינות רק בעת שימוש במשתנה **KeywordToken** או **IdentifierToken**.

הערה

כעת ברור מדוע ניתן להציב כמעט כל דבר במשתנה מסוג **object**. זכור ש-**object** הוא כינוי למחלקה **system.Object**, וכל המחלקות יורשות מהמחלקה **system.Object** באופן ישיר או עקיף.

שיטות new

אחת מהבעיות הקשות בתחום תכנות המחשבים היא להמציא שמות חדשים ובעלי משמעות למזהים (identifiers). אם הינך מגדיר שיטה עבור מחלקה, ואותה מחלקה משתייכת להיררכיית הורשה, במוקדם או במאוחר תנסה למחזר שם אשר נמצא כבר בשימוש של אחת המחלקות שנמצאות במעלה ההיררכיה. אם מחלקת בסיס ומחלקה נגזרת מכריזות במקרה על שתי שיטות בעלות חתימה זהה (חתימה של שיטה היא שם השיטה בשילוב של מספר הפרמטרים וסוגיהם), תופיע הודעת אזהרה בשלב הידור היישום. השיטה במחלקה הנגזרת מסתירה את השיטה במחלקת הבסיס אשר חתימתה זהה לשלה. לדוגמה, אם תהדר את הקוד הבא, המהדר יציג הודעת אזהרה על כך שהשיטה **IdentifierToken.Name** מסתירה את השיטה **Token.Name**:

```
class Token
{
   ...
   public string Name() { ... }
}

class IdentifierToken : Token
{
   ...
   public string Name() { ... }
}
```

למרות שהקוד יעבור הידור ויופעל, עליך להתחשב באזהרה זו. אם מחלקה אחרת הנגזרת מהמחלקה **IdentifierToken** תקרא לשיטה **Name** מתוך כוונה להפעיל את השיטה **Name** שהוגדרה במחלקה **Token**, השיטה שלמעשה תופעל תהיה **Name** מהמחלקה **IdentifierToken**, מכיוון שהיא מסתירה את **Token.Name**. לרוב, צירוף מקרים שכזה עשוי לגרום לבלבול רב, ולכן רצוי לשנות את שמות השיטות כדי להימנע מהתנגשויות כאלו. אולם, אם אתה בכל זאת רוצה ליצור שתי שיטות בעלות חתימה זהה, תוכל לבטל את האזהרה באמצעות מילת המפתח **new**, הנה כך:

```
class Token
{
   ...
   public string Name() { ... }
}

class IdentifierToken : Token
{
   ...
   new public string Name() { ... }
}
```

שימוש זה במילת המפתח **new** אינו משנה את העובדה ששתי המחלקות אינן קשורות זו לזו, ושאחת מהן עודנה מסתירה את השנייה. הפעולה גורמת לביטול האזהרה בלבד. מילת המפתח **new** מהווה למעשה דרך לומר למהדר "אני יודע מה אני עושה, תפסיק להזהיר אותי".

שיטות וירטואליות

בתכנות מוכוון עצמים נפוץ השימוש בייחוס של מחלקת בסיס אל אובייקטים נגזרים. בשפת C# ניתן לממש זאת באמצעות שיטות וירטואליות. **שיטה וירטואלית** (virtual method) מאפשרת לקרוא לשיטה המשויכת לסוג המוצב בפועל, במקום לקרוא לשיטה המשויכת לסוג מחלקת הבסיס.

לעיתים נרצה להסתיר את הדרך ליישום השיטה במחלקת הבסיס. ניקח לדוגמה את השיטה **ToString** שנמצאת ב-**System.Object**. מטרת השיטה **ToString** להמיר אובייקט למחרוזת המייצגת אותו. מכיוון ששיטה זו מאוד שימושית, היא חברה במחלקה **System.Object** ולפיכך כל המחלקות כוללות את השיטה **ToString**. אולם, כיצד יודעת הגרסה של **ToString** מהמחלקה **System.Object** כיצד להמיר מחלקה נגזרת למחרוזת? מחלקה מכילה מספר בלתי קבוע של שדות המכילים ערכים שונים, וכולם אמורים להוות חלק מהמחרוזת.

התשובה היא שהשיטה **ToString** שבמחלקה **System.Object** היא פשטנית למדי. כל מה שהיא יכולה לעשות זה להמיר אובייקט למחרוזת המייצגת את הסוג שלו, כגון "Token" או "IdentifierToken". פעולה זו אינה שימושית במיוחד. אם כך, מדוע ליצור שיטה שאינה שימושית? התשובה לשאלה זו קצת יותר מורכבת.

אין ספק ש**הרעיון** שביסוד השיטה **ToString** הוא טוב, כי בכל מחלקה כדאי שתהיה שיטה אשר מסוגלת להמיר אובייקטים למחרוזות. הבעיה היא במימוש הרעיון. למעשה, אינך אמור לקרוא לשיטה **ToString** אשר הוגדרה על ידי **System.Object**, אלא להשתמש בה כשומר מקום (place holder). עליך ליצור גרסה משלך למחלקה **ToString** בכל מחלקה שתגדיר, וכך לעקוף או לדרוס (override) את ברירת המחדל של המחלקה **System.Object**. הגרסה שנמצאת ב-**System.Object** משמשת כרשת ביטחון למקרים שבהם מחלקה מסוימת אינה מכילה שיטה **ToString** משלה. באופן זה ניתן לדעת בוודאות שניתן להפעיל את השיטה **ToString** על כל אובייקט ולקבל מחרוזת, ללא תלות בדרך מימוש השיטה.

שיטה אשר נוצרה בידיעה שיעקפו אותה נקראת **שיטה וירטואלית** (virtual method). חשוב להפריד בין הסתרת שיטה ועקיפת שיטה. **עקיפה** (או **דריסה**, overriding) היא מנגנון שאפשר דרכים שונות למימוש של אותה שיטה - השיטות השונות קשורות זו לזו מכיוון שכולן אמורות לבצע את אותה הפעולה. **הסתרה** (hiding) היא אמצעי להחלפת שיטה אחת בשיטה אחרת - לרוב השיטות אינן קשורות זו לזו וככל הנראה הן מבצעות פעולות שונות לחלוטין. עקיפה של שיטה היא אמצעי חשוב בתכנות, לעומת הסתרת שיטה שנובעת לרוב כתוצאה משגיאה.

ניתן לסמן שיטות כווירטואליות באמצעות מילת המפתח virtual. לדוגמה, השיטה ToString שבמחלקה System.Object מוגדרת כך:

```
namespace System
{
   class Object
   {
      public virtual string ToString()
      {
         ...
      }
      ...
   }
   ...
}
```

הערה

בדומה ל-++C, שיטה ב-#C אינה וירטואלית על פי ברירת מחדל. אולם בשפת Java כל השיטות הן וירטואליות, אלא אם כן נאמר אחרת.

שיטות וירטואליות ופולימורפיזם

למרות שהורשה פשוטה מספקת, קיימות סיבות לפעול בדרך שונה. אחד ממושגי המפתח של שפה מוכוונת עצמים הוא **רב-צורתיות** (polymorphism). אם הורשה נעשית בצורה נכונה, היא עוזרת בהשגת יתרונות רב-צורתיים בתוך מחלקות. ליתר דיוק, אפשר ליצור היררכיית מחלקות שניתן להתייחס אליהן באופן כמעט זהה.

שיטות וירטואליות מאפשרות לקרוא לגרסאות שונות של אותה שיטה בהתאם לסוג האובייקט אשר נקבע באופן דינמי בסביבת ההרצה (runtime). עיין במחלקות הבאות אשר מגדירות וריאציה של היררכיית היונקים שתוארה קודם לכן:

```
class Mammal
{
   ...
   public virtual string GetTypeName()
   {
      return “This is a mammal”;
   }
}
class Horse : Mammal
{
   ...
   public override string GetTypeName()
   {
      return “This is a horse”;
   }
}
class Whale : Mammal
{
   ...
   public override string GetTypeName ()
   {
      return “This is a whale”;
   }
}
class Kangaroo : Mammal
{
   ...
}
```

שים לב לשני דברים: ראשית, מילת המפתח **override** משמשת את השיטה **GetTypeName** (שנתאר מייד) במחלקות **Horse** ו-**Whale**; שנית, המחלקה **Kangaroo** אינה כוללת את השיטה **GetTypeName**.

```
Mammal m;
Horse h = new Horse();
Whale w = new Whale();
Kangaroo k = new Kangaroo();

m = h;
Console.WriteLine(m.GetTypeName()); // Horse
m = w;
Console.WriteLine(m.GetTypeName()); // Whale
m = k;
Console.WriteLine(m.GetTypeName()); // Kangaroo
```

מה יהיה הפלט של שלושת משפטי **Console.WriteLine** השונים? במבט ראשון ניתן לשער ששלושת המשפטים ידפיסו "This is a mammal" על המסך, מכיוון שכל אחד מהם מפעיל את השיטה **GetTypeName** על המשתנה m שהוא מסוג **Mammal**. אולם, במקרה הראשון ניתן לראות ש-m מתייחס למעשה ל-**Horse** (מותר לך להציב **Horse** ב-**Mammal**. מכיוון שהמחלקה **Horse** נגזרת מהמחלקה **Mammal** - כי כל הסוסים הם יונקים). מכיוון שהשיטה **GetTypeName** מוגדרת כווירטואלית, סביבת ההרצה מניחה

שעליה לקרוא לשיטה **Horse.GetTypeName**, ולכן תופיע ההודעה "This is a horse". מאותה סיבה המשפט **Console.WriteLine** השני יגרום להצגת ההודעה "This is a whale". המשפט השלישי מפעיל את השיטה **Console.WriteLine** על האובייקט **Kangaroo**. אולם במחלקה **Kangaroo** אין שיטה **GetTypeName**, ולכן מופעלת שיטת ברירת המחדל מהמחלקה **Mammal**, ועל המסך תופיע גם כן המחרוזת "This is a mammal".

תופעה זו, שבה משפט בודד יכול להפעיל מספר שיטות, נקראת **ריבוי-צורות** (polymorphism).

שיטות עקיפה

במקרה ששיטה הוכרזה כווירטואלית במחלקת הבסיס, ניתן להשתמש במילת המפתח **override** (כלומר, עקיפה או דריסה) כדי להכריז על חלופה שונה ליישום של אותה שיטה. לדוגמה:

```
class IdentifierToken : Token
{
   ...
   public override string Name() { ... }
}
```

עליך לקיים מספר כללים חשובים בעת הכרזה על שיטות פולימורפיות באמצעות מילות המפתח **virtual** ו-**override**:

- אסור להכריז על שיטה פרטית באמצעות מילות המפתח **virtual** או-**override**. אם תנסה לעשות זאת, תתקבל הודעת שגיאה בשלב ההידור. שיטה פרטית כשמה כן היא (private).
- שתי השיטות צריכות להיות זהות זו לזו. כלומר, הן חייבות להיות בעלות אותו שם ואותם סוגי פרמטרים ועליהן להחזיר אותו ערך.
- שתי השיטות צריכות להיות בעלות נגישות זהה. לדוגמה, אם אחת מהן היא ציבורית (public), גם השנייה חייבת להיות ציבורית.
- ניתן לעקוף שיטות וירטואליות בלבד. אם השיטה שבמחלקת הבסיס אינה וירטואלית, אולם תנסה לעקוף אותה בכל זאת, תופיע הודעת שגיאה בשלב ההידור. יש בכך הגיון, כי ההחלטה אם השיטה יכולה להיעקף או לא צריכה להתקבל במסגרת מחלקת הבסיס.
- אם ההכרזה על השיטה במחלקה הנגזרת מתבצעת ללא מילת המפתח override, השיטה לא תעקוף את השיטה שבמחלקת הבסיס. הדבר יגרום להודעת אזהרה בשלב ההידור, אשר ניתן להשתיקה באמצעות מילת המפתח new באופן שהוצג מוקדם יותר בפרק.
- השיטה **override** היא תמיד וירטואלית, ולכן ניתן לעקוף אותה ממחלקה שנגזרת מהמחלקה שבה היא נמצאת. עם זאת, לא ניתן (וגם אין צורך) להכריז על השיטה **override** כווירטואלית באמצעות מילת המפתח **virtual**.

הגנה על הגישה

מילות המפתח **public** ו-**private** יוצרות שני מצבי קיצון של נגישות: שדות ושיטות ציבוריים של מחלקה כלשהי נגישים לכל, בעוד ששדות ושיטות פרטיים נגישים אך ורק למחלקה עצמה.

שני מצבי הקיצון הללו עונים לכל הדרישות כאשר המחלקות מבודדות. אולם, כפי שיודעים מתכנתים עם ניסיון בתכנות מונחה עצמים, מחלקות מבודדות אינן מסוגלות לפתור בעיות מורכבות. הורשה היא דרך בעלת עוצמה לחיבור בין מחלקות, ואין ספק שיש קשר מיוחד וקרוב מאוד בין מחלקה נגזרת ומחלקת הבסיס שלה. לעיתים קרובות יהיה זה שימושי ביותר אם מחלקת הבסיס תוכל לאפשר למחלקות הנגזרות ממנה גישה לחלק מהחברים (members) בה, ובאותו זמן תחסום את הגישה לאותם חברים בפני מחלקות שאינן חלק מן ההיררכיה. במצבים כאלה ניתן להשתמש במילת המפתח **protected** (מוגן) כדי לסמן חברים במחלקה כמוגנים:

- למחלקה נגזרת יש גישה לחבר מוגן במחלקת הבסיס. במילים אחרות, מתוך המחלקה הנגזרת, חבר מוגן במחלקת הבסיס הוא לכאורה ציבורי.
- אם המחלקה איננה נגזרת, אין לה גישה לחברי מחלקה מוגנים. במילים אחרות, מתוך מחלקה שאיננה נגזרת, חבר מחלקה מוגן הוא לכאורה פרטי.

שפת #C נותנת למתכנתים חופש מוחלט להכריז על שיטות ושדות כמוגנים. אולם, מרבית ההנחיות העוסקות בתכנות מונחה עצמים ממליצות תמיד להכריז על **שדות** כפרטיים. שדות ציבוריים פוגעים בכימוס (encapsulation), מכיוון שלכל המשתמשים במחלקה יש גישה ישירה ובלתי מוגבלת לשדות. שדות מוגנים משמרים את הכימוס בפני מחלקות שמהן לא מתאפשרת גישה לשדות המוגנים. עם זאת, שדות מוגנים עדין עשויים לפגוע בכימוס כאשר מדובר במחלקות היורשות מהמחלקה שהם נמצאים בה.

הערה

גישה לחבר מוגן במחלקת הבסיס מתאפשרת מהמחלקה הנגזרת, וגם ממחלקה הנגזרת מהמחלקה הנגזרת. חבר מוגן במחלקת הבסיס נותר בעל גישה מוגנת במחלקה הנגזרת והינו נגיש לכל המחלקות הנגזרות בהמשך.

יצירת ממשקים

ירושה ממחלקה היא מנגנון בעל עוצמה רבה, אך העוצמה האמיתית של ההורשה נובעת מהורשת ממשק. הממשק מאפשר להפריד לחלוטין בין שם השיטה לבין מימושה.

לדוגמה, נניח שברצונך להגדיר מחלקת אוסף (collection class) חדשה אשר מאפשרת לאחסן אובייקטים בסדר מסוים. בעת הגדרת מחלקת האוסף, אינך רוצה להגביל את סוג האובייקטים שביכולתה לאחסן, אך ברצונך ליצור דרך לארגון האובייקטים לפי הסדר הרצוי. השאלה היא כיצד לבנות שיטה אשר מסדרת אובייקטים מסוג לא ידוע בעת כתיבת מחלקת האוסף. במבט ראשון בעיה זו מזכירה את הבעיה שהוצגה קודם עם השיטה **ToString**, אשר ניתן לפתור אותה באמצעות הכרזה על שיטה וירטואלית הניתנת לעקיפה על ידי שיטות אחרות. אולם כאן לפנינו מצב שונה. אין בהכרח כל קשר של הורשה בין מחלקת האוסף לבין האובייקטים המאוחסנים בה ולכן, שיטה וירטואלית אינה הפתרון לבעיה זו. הפתרון הוא לציין שכל האובייקטים באוסף חייבים לכלול שיטה המאפשרת להשוות אותם זה לזה, כגון השיטה **CompareTo** המוצגת להלן:

```
int CompareTo(object obj)
{
   // return 0 if this instance is equal to obj
   // return < 0 if this instance is less than obj
   // return > 0 if this instance is greater than obj
   ...
}
```

מחלקת האוסף תוכל להשתמש בשיטה זו כדי לסדר את הפריטים המאוחסנים בה.

ניתן להגדיר ממשק הכולל את השיטה הזו, ולציין שמחלקת האוסף מאחסנת אך ורק מחלקות אשר מממשות את הממשק הזה. מנגנון זה מבטיח שתוכל להפעיל את השיטה **CompareTo** על כל אחד מהאובייקטים שבמחלקה כדי לסדרם בסדר הרצוי.

ממשקים מאפשרים ליצור הפרדה אמיתית בין ה"מה" ל"איך". הממשק אומר מהו שם השיטה, מהו הסוג שהיא מחזירה ומהם הפרמטרים שהיא מקבלת. הממשק אינו עוסק באופן מימוש השיטה, אך מייצג את הדרך שבה אתה רוצה להשתמש באובייקט, ולא איך זה נעשה בכל רגע נתון.

מדוע להשתמש בממשקים?

יש יתרונות לשימוש בממשקים. ניתן להשתמש בממשקים כאמצעי לספק תכונות הורשה מסוימות באמצעות מבנים מוגדרים. כמו כן, ניתן לממש ממשקים מרובים במחלקה אחת, ובכך להשיג את הפונקציונליות שלא ניתן להשיגה עם מחלקה מופשטת.

אחד היתרונות הגדולים לשימוש בממשקים הוא בכך שניתן להוסיף למחלקה תכונות שאינן ניתנות להוספה בדרך אחרת. אם מוסיפים אותן התכונות למחלקות אחרות, מתחילים להניח הנחות אודות הפונקציונליות של המחלקה. למעשה, השימוש בממשקים מונע ממך להניח הנחות.

יתרון נוסף לשימוש בממשק במקום במחלקה הוא בכך שמאלצים את המחלקה החדשה לממש את כל התכונות המוגדרות בממשק. אם יורשים ממחלקת בסיס המכילה חברים וירטואליים, לא חייבים לממש קוד עבורם. מצב כזה חושף לשגיאות את המחלקה החדשה ואת התוכניות המשתמשות בה.

תחביר של ממשק

מכריזים על ממשק באמצעות מילת המפתח **interface** ולא במילות המפתח **class** או **struct**. בתוך הממשק, הגדרת השיטות נעשית באופן זהה לזה של מחלקות ומבנים, מלבד העובדה שלא ניתן לשנות את הגישה (גישה ציבורית, פרטית או מוגנת), ושאת גוף השיטה מחליף נקודה-פסיק. הנה דוגמה:

```
interface IComparable
{
   int CompareTo(object obj);
}
```

טיפ

ב- Microsoft .NET Framework documentation מומלץ להוסיף I גדולה לפני שם הממשק. המלצה זו היא האיזכור האחרון לשיטת הסימון ההונגרית ב-C#. ראוי לציין שבמרחב השמות System, מוגדר כבר הממשק **IComparable**.

מגבלות הממשק

הכי חשוב לזכור שהממשק לעולם אינו מכיל **מימושים** (implementations). המגבלות הבאות נובעות ישירות מעובדה זו:

- ממשק אינו יכול להכיל שדות, גם לא סטטיים. שדה מממש למעשה תכונה של אובייקט.
- ממשק אינו יכול להכיל בנאים. בנאי מכיל משפטים המשמשים לאתחול השדות באובייקט, וממשק אינו מכיל שדות!
- ממשק אינו יכול להכיל מפרקים (destructors). מפרק מכיל משפטים המשמשים לחיסול מופע של אובייקט (על מפרקים נלמד בפרק 13).
- לא ניתן לרשת ממשק ממבנה או ממחלקה. מבנים ומחלקות מכילים מימושים. אם הממשק היה יכול לרשת מכל אחד מהם, הוא היה יורש חלק מהמימושים.

מימוש הממשק

כדי לממש (implement) ממשק, עליך להכריז על מחלקה או מבנה אשר יורשים מהממשק וממשים את **כל** השיטות שלו. לדוגמה, נניח שאתה מגדיר את ההיררכיה **Token**, אולם עליך לציין שכל המחלקות שבהיררכיה כוללות שיטה בשם **Name** אשר מחזירה את שם היחידה הלקסיקלית (token) בתור מחרוזת. תוכל להגדיר ממשק **IToken** אשר מכיל את השיטה הזו:

```
interface IToken
{
   string Name();
}
```

את הממשק הזה תוכל לממש במחלקה **Token**:

```
class Token : IToken
{
   ...
   string IToken.Name()
   {
      ...
   }
}
```

בעת מימוש של ממשק, עליך לוודא שכל אחת מהשיטות תואמת במדויק לשיטה המקבילה לה בממשק, על פי ההנחיות הבאות:

- שמות השיטות והערכים שהן מחזירות צריכות להיות זהים לחלוטין.
- כל הפרמטרים וגם מילות המפתח **ref** ו-**out**, אולם לא מילת המפתח **params**, צריכים להתאים בדיוק.
- לפני שם השיטה יש להציב את שם הממשק. פעולה זו ידועה בשם **מימוש ממשק מפורש** (**explicit interface implementation**), ורצוי לאמץ אותה כהרגל.
- אם אתה משתמש במימוש ממשק מפורש, לא ניתן להוסיף לשיטה וסת גישה. שיטות אשר מממשות ממשק לעולם תהיינה ציבוריות.

הבדלים בין הגדרות הממשק ובין המימוש שלו יגרמו לכך שהמחלקה לא תהודר.

היתרונות של מימוש ממשק מפורש

במבט ראשון נראה שמימוש מפורש של ממשק הינו פעולה מסורבלת, אך יש לשיטה זו מספר יתרונות שמאפשרים לכתוב קוד ברור יותר, גמיש יותר וצפוי יותר.

ניתן לממש שיטה מבלי לציין באופן מפורש את שם הממשק, אולם הדבר עשוי להוביל לשינויים בהתנהגות המימוש. חלק מהשינויים הללו עשוי לגרום לבלבול. לדוגמה, לא ניתן להגדיר כו**וירטואלית** שיטה שמומשה באמצעות מימוש ממשק מפורש, אולם השמטה של שם הממשק תאפשר התנהגות כזו.

מימוש ממשק מפורש מונע בלבול בין שיטות של ממשקים שונים, אך בעלות אותו השם, אותו ערך מוחזר ואותם פרמטרים. השיטות אשר מממשות את הממשק הן ציבוריות, אולם רק בעיני הממשק (נלמד כיצד לעשות זאת בהמשך הפרק). ללא מימוש ממשק מפורש, לא היה ניתן להבחין בין שיטות אשר מממשות חלקים של ממשקים שונים, במקרים שבהם הממשקים מכילים שיטות בעלות אותו שם, אותו סוג ערך מוחזר ואותם פרמטרים. המלצתנו היא **לממש תמיד את הממשק באופן מפורש**, אלא אם כן לא ניתן לעשות זאת.

מחלקה יכולה להרחיב מחלקה אחרת ולממש את הממשק בו-זמנית. במקרה הבא, שפת #C (בדומה ל-Java, למשל) אינה מבדילה בין מחלקת הבסיס לבין הממשק באמצעות מילת המפתח as. שפת #C עושה שימוש בסימון מיקום (positional notation). ראשונה תופיע מחלקת הבסיס, לאחר מכן פסיק, ולבסוף הממשק. הנה דוגמה:

```
interface IToken
{
   ...
}

class DefaultTokenImpl
{
   ...
}

class IdentifierToken : DefaultTokenImpl , IToken
{
   ...
}
```

הפניה למחלקה באמצעות הממשק שלה

ניתן להפנות אל אובייקט באמצעות משתנה שמוגדר כמחלקה שנמצאת גבוה יותר בהיררכיה. כך גם ניתן להפנות אל אובייקט באמצעות משתנה שמוגדר כממשק שהמימוש שלו נעשה על ידי מחלקה. נעזר שוב בדוגמה הקודמת. ניתן להפנות לאובייקט מסוג **IdentifierToken** באמצעות משתנה מסוג **IToken** באופן הבא:

```
IdentifierToken it = new IdentifierToken();
IToken iTok = it; // legal
```

שיטה זו שימושית, מכיוון שהיא מאפשרת להגדיר שיטות אשר יכולות לקבל סוגים שונים בתור פרמטרים, כל עוד יש בסוגים אלה מימוש של ממשק נתון. לדוגמה, השיטה **Process** שלהלן, יכולה לקבל כל ארגומנט אשר מממש את הממשק **IToken**.

```
void Process(IToken iToken)
{
   ...
}
```

שים לב שניתן להפעיל רק שיטות שניתן לראותן מתוך הממשק, כאשר מפנים אל אובייקט בדרך זו.

Classes לעומת Interfaces

יש דמיון בין **ממשק** (interface) לבין **מחלקה מופשטת טהורה** (pure abstract class). ממשק שונה ממחלקה במספר היבטים:

1. ממשק אינו מספק קוד מימוש כלשהו. הדבר נעשה על ידי המחלקות המממשות אותו. על ממשק נאמר שהוא מספק מפרט, או קו מנחה, למה שיתרחש, אך לא את הפרטים.
2. ממשק שונה ממחלקה בכך שכל חבריו נחשבים לציבוריים (public). אם מנסים להכריז על חבר ממשק עם מציין טווח הכרה אחר, מקבלים שגיאה.
3. ממשקים כוללים אך ורק שיטות, מאפיינים, אירועים וסדרנים (indexers). הם אינם יכולים להכיל משתנים חברים, בנאים, מפרקים וחברים סטטיים.

עבודה עם ממשקים אחדים

למחלקה יכולה להיות מחלקת בסיס אחת בלבד, אך היא יכולה ליצור מספר בלתי מוגבל של ממשקים. המחלקה צריכה ליצור את כל השיטות שהיא יורשת מכל הממשקים שלה. ממשק אינו יכול לרשת ממחלקה מכל סוג שהוא, אולם הוא יכול להוריש לממשקים אחרים.

אם ממשק, מבנה או מחלקה יורשים משני ממשקים או יותר, עליך לכתוב את שמות הממשקים ברשימה המופרדת על ידי פסיקים. אם למחלקה כלשהי יש גם מחלקת בסיס, עליך לרשום את שמות הממשקים **אחרי** מחלקת הבסיס. לדוגמה:

```
class IdentifierToken : DefaultTokenImpl, IToken, IVisitable
{
   ...
}
```

גזירת ממשקים חדשים מממשקים קיימים

ניתן לגזור ממשק מממשק אחר. הורשה של ממשקים מתבצעת בדרך דומה להורשת מחלקות. קטע הקוד הבא מדגים איך אפשר להרחיב את הממשק **IShape** שראינו בדוגמה קודמת:

```
public interface IShape
{
   long Area();
   long Circumference();
   int Sides{ get; set; };
}

interface I3DShape : IShape
{
   int Depth { get; set; }
}
```

הממשק **I3DShape** מכיל את כל החברים של הממשק **IShape** וכן כל חבר חדש שהוא עצמו מוסיף. ניתן להשתמש ב-**I3DShape** כפי שמשתמשים בכל ממשק אחר. חברי הממשק יהיו **Area**, **Circumference**, **Sides** ו-**Depth**.

מחלקות מופשטות

מספר רב של מחלקות שונות יכולות לממש את הממשק **IToken**, מחלקה עבור כל סוג של יחידה לקסיקלית שבקובץ המקור של #C: **IdentifierToken**, **KeywordToken**, **LiteralToken**, **OperatorToken** ו-**PanctuatorToken** (ייתכן שתהיינה גם מחלקות עבור הערות ועבור תווים לבנים). במצבים כאלה, מקובל מאוד שחלקים מהמחלקות הנגזרות ישתפו ביניהם קטעים משותפים בממשק. לדוגמה, קל לראות את המשותף בין שתי המחלקות הבאות:

```
class IdentifierToken : IToken
{
   public IdentifierToken(string name)
   {
      this.name = name;
   }
   public virtual string Name()
   {
      return name;
   }
   ...
   private string name;
}

class StringLiteralToken : IToken
{
   public StringLiteralToken(string name)
   {
      this.name = name;
   }
   public virtual string Name()
   {
      return name;
   }
   ...
   private string name;
}
```

כדאי לראות בחזרה על קטעי קוד אות אזהרה. עליך לשנות את הקוד כדי להימנע מחזרות, וכדי להפחית את עלויות האחזקה של התוכנית. יש דרך נכונה לבצע את השינויים הללו וצריך לנהוג לפיה. טעות נפוצה במקרים כאלה היא להעביר את החלק אשר חוזר על עצמו רמה אחת למעלה, אל הממשק. אין זה הדבר הנכון לעשות, מכיוון שכדי ליישם זאת עלינו לשנות את הממשק למחלקה (מכיוון שממשק אינו יכול להכיל מימושים). הדרך הנכונה להימנע מחזרות היא להפוך את המקטע אשר חוזר על עצמו למחלקה חדשה שניצור במיוחד למטרה זו. לדוגמה:

```
class DefaultTokenImpl
{
   public DefaultTokenImpl(string name)
   {
      this.name = name;
   }
   public string Name()
   {
      return name;
   }
   private string name;
}

class IdentifierToken : DefaultTokenImpl, IToken
{
   public IdentifierToken(string name): base(name)
   {
   }
```

```
    ...
}
class StringLiteralToken : DefaultTokenImpl, IToken
{
    public StringLiteralToken(string name): base(name)
    {
    }
    ...
}
```

זהו פתרון טוב. עם זאת, נותרה בעיה אחת: ניתן ליצור מופע של המחלקה **DefaultTokenImpl** למרות שאין בכך שמץ של היגיון. מחלקה זו קיימת בכדי לאפשר את המימוש של ברירת מחדל משותפת. יעודה היחיד הוא שיהיה ניתן לרשת ממנה. כלומר, המחלקה **DefaultTokenImpl** הינה הפשטה של פעולה משותפת ולא ישות בפני עצמה.

הערה

אם השימוש במחלקה **DefaultTokenImpl** מבלבל אותך, תאר לעצמך את הדוגמה של היונקים שהוצגה בתחילת הפרק. **Mammal** היא דוגמה קלאסית למחלקה **מופשטת**. בעולם האמיתי, יתכן שתראה **סוסים**, **לוויתנים** ו**קנגרו** רצים, קופצים ושוחים, אולם לעולם לא תראה "יונק" עושה את הדברים הללו. יונק הוא מושג מופשט שנועד לסווג את החיות הללו.

כדי להכריז שלא ניתן ליצור מופע של מחלקה מסוימת, עליך להכריז עליה **כמופשטת** (abstract) באמצעות מילת המפתח **abstract**, כמובן. לדוגמה:

```
abstract class DefaultTokenImpl
{
    public DefaultTokenImpl(string name)
    {
        this.name = name;
    }
    public string Name()
    {
        return name;
    }
    private string name;
}
```

שים לב שהמחלקה החדשה **DefaultTokenImpl** אינה מממשת את הממשק **IToken**. היא הייתה יכולה לעשות זאת, אולם הממשק IToken אינו מתאים למטרתה. מחלקה מופשטת משמשת אך ורק למימוש משותף, בעוד שמטרת הממשק היא רק השימוש. רצוי ליצור הפרדה בין שני ההיבטים הללו ולאפשר למחלקות שאינן מופשטות (כגון **StringLiteralToken**) לקבוע כיצד הן יממשו את הממשקים שלהן:

- הן יכולות לרשת מ-**DefaultTokenImpl** ומ-**IToken**, ובמקרה זה **DefaultTokenImpl.Name** תהפוך למימוש של **IToken.Name**. חשוב לשים לב שמשמעות הדבר היא שהמחלקה **DefaultTokenImpl.Name** חייבת להיות ציבורית. ניתן להכריז על הבנאי של **DefaultTokenImpl** כמוגן (protected), אולם השיטה **Name** חייבת להישאר ציבורית כדי שתוכל לממש את **IToken.Name** במחלקה נגזרת.

- מחלקות שאינן יורשות מ-**DefaultTokenImpl** צריכות לממש בעצמן את **IToken.Name**.

מחלקות חתומות

השימוש בהורשה אינו תמיד פשוט, ולעיתים אף צריך לתכנן בקפידה מראש כיצד לעשות זאת. בעת יצירת ממשק או מחלקה מופשטת, אתה כותב באופן מודע משהו שירשו ממנו בעתיד. הבעיה היא שקשה מאוד לחזות את העתיד. כדי לעשות זאת צריך מיומנות, מאמץ רב והכרה של הבעיה שברצונך לפתור, ורק כך יהיה ניתן לעצב היררכיה גמישה וקלה לשימוש של ממשקים, מחלקות מופשטות ומחלקות רגילות. במילים אחרות, אם לא תעצב באופן מודע את המחלקה אשר אמורה לשמש בתור מחלקת בסיס, קרוב לוודאי שהיא לא תתפקד כמחלקת בסיס יעילה. למרבה המזל, שפת C# מאפשרת שימוש במילת המפתח **sealed** (**חתום**) כדי להגדיר מחלקה כזו. כך ניתן למנוע מצב שבו מחלקה שלא אמורה לשמש כמחלקת בסיס תשמש ככזו בטעות. לדוגמה:

```
sealed class LiteralToken : DefaultTokenImpl, IToken
{
    ...
}
```

אם מחלקה תנסה להשתמש במחלקה **LiteralToken** בתור מחלקת בסיס, הדבר יגרום לשגיאה בשלב ההידור. במחלקה **חתומה** (sealed) לא ניתן להכריז על שיטות וירטואליות. המטרה היחידה של מילת המפתח **virtual** היא להכריז כי זהו המימוש **הראשון** של השיטה שבכוונתך לעקוף (override) במחלקה נגזרת, אולם ממילא לא ניתן לגזור מחלקות אחרות ממחלקה חתומה.

הערה

מבנה (struct) לעולם יהיה חתום. אי-אפשר לגזור ממבנה.

שיטות חתומות

ניתן להשתמש במילת המפתח **sealed** כדי להכריז על שיטה יחידה כחתומה. משמעות הדבר היא שמחלקה נגזרת אינה יכולה לעקוף את השיטה החתומה. ניתן לסכם את ייעודי מילות המפתח **interface**, **virtual**, **override** ו-**sealed** באופן הבא:

- הממשק מכריז על **השם** של השיטה.
- השיטה הווירטואלית היא המימוש **הראשון** של השיטה.
- השיטה העוקפת היא מימוש **נוסף** של השיטה.
- השיטה החתומה היא המימוש **האחרון** של השיטה.

הרחבה של היררכיית ההורשה

בתרגיל הבא תבחן היררכיה קטנה של ממשקים ומחלקות המרכיבים יחדיו מערכת פשוטה למדיי. מערכת זו היא למעשה יישום של Microsoft Windows המדמה קריאת קובץ מקור של #C ומיון תוכנו ליחידות לקסיקליות (מזהים, מילות מפתח, אופרטורים וכו'). המערכת כוללת מנגנון המאפשר "לבקר" (visit) בכל אחת מהיחידות הלקסיקליות (או מטבעות, tokens), ולבצע מטלות שונות. המערכת כוללת בין השאר:

- **מחלקה מבקרת מציגה** - מציגה את קובץ המקור בתיבת טקסט עשיר.
- **מחלקה מבקרת מדפיסה** - ממירה טאבים לרווחים ומיישרת סוגריים בצורה נכונה.
- **מחלקה מבקרת מאייתת** - מוודאת איות נכון של המזהים.
- **מחלקה מבקרת מנחה** (guidline) - בודקת שהמזהים הציבוריים מתחילים באות רישית (גדולה) ושהממשקים מתחילים באות I רישית.
- **מחלקה מבקרת לבדיקת מורכבות** - בודקת את עומק הסוגריים שבקוד.
- **מחלקה מבקרת סופרת** - סופרת את מספר השורות בכל שיטה, מספר החברים בכל מחלקה ומספר השורות בכל קובץ מקור.

היררכיית ההורשה ויעודה

הבה נראה כיצד זה פועל:

1. הפעל את Microsoft Visual Studio 2005.
2. פתח את הפרוייקט **Tokenizer**, שבתיקייה My Documents\Microsoft Press\Visual CSharp Step by Step\Chapter 12\Tokenizer.
3. הצג את קובץ המקור SourceFile.cs בחלון Code and Text Editor.

המחלקה **SourceFile** מכילה שדה מערך פרטי בשם **tokens**:

```
private IVisitableToken[] tokens =
{
   new KeywordToken("using"),
   new WhitespaceToken(" "),
   new IdentifierToken("System"),
   new PunctuatorToken(";"),
   ...
};
```

המערך **tokens** מכיל רצף של אובייקטים שכולם כוללים את הממשק **IVisitableToken**. 'מטבעות' אלו מדמות יחדיו את המטבעות של קובץ מקור "hello, world" פשוט (גרסה מלאה של פרויקט זה הייתה מפרקת לגורמים קובץ מקור, ובונה את סדרת המטבעות באופן דינמי). המחלקה SourceFile מכילה גם שיטה ציבורית בשם **Accept**.

השיטה Accept מקבלת פרמטר יחיד מסוג **ITokenVisitor**. גוף השיטה **Accept** מבצע דפדוף בין המטבעות שבמערך, וקורא לשיטה **Accept** של כל אחד מהם. השיטה **Accept** של כל אחד מהמטבעות מעבדת את המטבע בהתאם לסוג שלו:

```
public void Accept(ITokenVisitor visitor)
{
   foreach (IVisitableToken token in tokens)
   {
      token.Accept(visitor);
   }
}
```

באופן זה, הפרמטר **visitor** 'מבקר' בכל אחד מהמטבעות.

.4 הצג את קובץ המקור IVisitableToken.cs בחלון Code and Text Editor.

הממשק **IVisitableToken** יורש משני ממשקים אחרים: הממשק **IVisitable** והממשק **IToken**:

```
interface IVisitableToken : IVisitable, IToken
{
}
```

.5 הצג את קובץ המקור IVisitable.cs בחלון Code and Text Editor.

הממשק **IVisitable** מכריז על שיטה אחת בשם **Accept**:

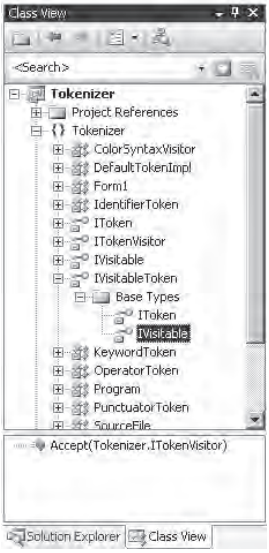
```
interface IVisitable
{
   void Accept(ITokenVisitor visitor);
}
```

ניתן לקבל גישה לכל אחד מהאובייקטים במערך המטבעות שבמחלקה **SourceFile** באמצעות הממשק **IVisitableToken**. ממשק זה יורש את השיטה **Accept**, וכל Token מממש את השיטה **Accept**.

.6 לחץ על הכרטיסייה Class View שמתחת לחלון Solution Explorer (אם הכרטיסייה Class View איננה מוצגת, תוכל לבחור בתפריט View את Class View).

חלון זה מציג את מרחבי השמות, המחלקות והממשקים המוגדרים בפרויקט.

.7 הרחב את הפרויקט **Tokenizer**, ולאחר מכן הרחב גם את מרחב השמות **Tokenizer {}**. המחלקות והממשקים במרחב שמות זה יופיעו ברשימה. שים לב לסמלים השונים המבדילים בין מרחבי השמות והמחלקות. הרחב את הממשק **IVisitableToken**, ולאחר מכן הרחב את הצומת Base Types. הממשקים שכלולים בממשק **IVisitableToken** (**IToken** ו-**IVisitable**) יופיעו כלהלן:



8. בחלון Class View, לחץ לחיצה ימנית על המחלקה **IdentifierToken** ומתפריט הקיצור בחר Go To Definition, כדי להציג אותה בחלון Code and Text Editor (בפועל היא נמצאת בקובץ SourceFile.cs).

המחלקה **IdentifierToken** יורשת מהמחלקה המופשטת **DefaultTokenImpl** ומהממשק **IVisitableToken**. המחלקה מממשת את השיטה **Accept** באופן הבא:

```
void IVisitable.Accept(ITokenVisitor visitor)
{
   visitor.VisitIdentifier(this.ToString());
}
```

למחלקות Token האחרות בקובץ זה יש תבנית דומה.

9. בחלון Class View, לחץ לחיצה ימנית על הממשק **ITokenVisitor** ומתפריט הקיצור בחר Go To Definition. הפעולה תגרום להצגת קובץ המקור SourceFile.cs בחלון Code and Text Editor.

הממשק **ITokenVisitor** מכיל שיטה עבור כל סוג של מטבע. היררכיה זו של ממשקים, מחלקות מופשטות ומחלקות, מאפשרת ליצור מחלקה אשר מממשת את הממשק **ITokenVisitor**, ליצור מופע של מחלקה זו ולהעביר את האובייקט בתור פרמטר לשיטה **Accept** של המחלקה **SourceFile**. הנה דוגמה:

```
class MyVisitor : ITokenVisitor
{
   public void VisitIdentifier(string token)
   {
      ...
   }
   public void VisitKeyword(string token)
   {
```

```
        ...
    }
    ...
    static void Main()
    {
        SourceFile source = new SourceFile();
        MyVisitor visitor = new MyVisitor();
        source.Accept(visitor);
    }
}
```

כתוצאה, כל 'מטבע' בקובץ המקור יקרא לשיטה המתאימה באובייקט המבקר בו. באפשרותך ליצור מספר מחלקות מבקרות שונות המסוגלות לבצע מספר פעולות שונות עם המטבע שהן מבקרות בו.

בתרגיל הבא עליך ליצור מחלקה הנגזרת מממשק מסגרת מרכזי, אשר מימושו מציג את המטבעות של קובץ המקור בתיבת טקסט עשיר בתחביר צבעוני (למשל, מילות מפתח בכחול) באמצעות מנגנון ה"ביקורים".

כתוב את המחלקה ColorSyntaxVisitor

1. ב- Solution Explorer (לחץ על הכרטיסייה Solution Explorer שבחלון Class View). לחץ לחיצה כפולה על Form1.cs כדי להציג את הטופס Color Syntax בחלון Designer View.

קובץ זה מכיל לחצן Open המשמש לפתיחת הקובץ אשר יפורק למטבעות (tokenized), וגם תיבת טקסט עשיר להצגת המטבעות. תיבת טקסט עשיר דומה לתיבת טקסט רגילה, פרט לעובדה שהיא יכולה להציג תוכן בפורמטים שונים ולא טקסט פשוט בלבד.

תיבת הטקסט העשיר שבמרכז הטופס נקראת **codeText**, והלחצן נקרא **Open**.

2. לחץ לחיצה ימנית על הטופס ומתפריט הקיצור בחר View Code, כדי להציג את הקוד בחלון Code and Text Editor.

3. אתר את השיטה **Open_Click**.

שיטה זו מופעלת כאשר המשתמש לוחץ Open. עליך לממש את השיטה כדי שתציג את המטבעות שהוגדרו במחלקה **SourceFile** שבתיבת הטקסט העשיר, באמצעות האובייקט **ColorSyntaxVisitor**. ערוך את השיטה **Open_Click** באופן הבא:

```
private void Open_Click(object sender, EventArgs e)
{
   SourceFile source = new SourceFile();
   ColorSyntaxVisitor visitor = new ColorSyntaxVisitor(codeText);
   source.Accept(visitor);
}
```

עליך לזכור שהשיטה **Accept** מהמחלקה **SourceFile** מדפדפת בכל המטבעות, ומעבדת כל אחד מהם באמצעות מבקר (visitor) מתאים. במקרה זה, המבקר הוא האובייקט **ColorSyntaxVisitor**, הצובע כל מטבע בצבע מסוים.

4. פתח את קובץ המקור ColorSyntaxVisitor.cs בחלון Code and Text Editor.

המחלקה **ColorSyntaxVisitor** מומשה באופן חלקי. היא מימשה את הממשק **ITokenVisitor** והיא כבר מכילה שני שדות ובנאי כדי לאתחל הפניה בשם **target** אל תיבת הטקסט העשיר אשר משמשת להצגת מטבעות. משימתך היא לממש את השיטות שמוריש הממשק **ITokenVisitor**, ולאחר מכן להציג את המטבעות בתיבת הטקסט העשיר. במחלקה **ColorSyntaxVisitor**, יש להתייחס לתיבת הטקסט העשיר באמצעות המשתנה **target**.

5. בחלון Code and Text Editor, הוסף את השיטה **Write** למחלקה **ColorSyntaxVisitor**, באופן הבא. הקפד לעשות בדיוק כך:

```
private void Write(string token, Color color)
{
   target.AppendText(token);
   target.Select(this.index, this.index + token.Length);
   this.index += token.Length;
   target.SelectionColor = color;
}
```

קוד זה מספח כל מטבע אל תיבת הטקסט העשיר, בעל צבע מסוים. השיטה **Select** ששייכת לתיבת הטקסט העשיר בוחרת בבלוק של טקסט המסומן בנקודת התחלה ונקודת סיום. כעת ניתן לצבוע את הטקסט המסומן באמצעות הגדרת המאפיין **SelectionColor**. המשתנה **index** עוקב אחר נקודת הסיום הנוכחית של הטקסט בתיבת הטקסט העשיר, והוא מעודכן בכל פעם שמסופח טקסט חדש לתיבה.

כל אחת מהשיטות ה"מבקרות" השונות תקרא לשיטה Write הזו, כדי להציג את תוצאותיה.

6. בחלון Code and Text Editor הוסף את השיטות המממשות את הממשק **ITokenVisitor** בתוך המחלקה **ColorSyntaxVisitor**. עליך להשתמש ב-**Color.Blue** עבור מילות מפתח, **Color.Green** עבור **StringLiterals**, ו-**Color.Black** עבור כל השיטות האחרות (**Color** הוא מבנה (struct) שהוגדר במרחב השמות **System.Drawing**). שים לב שקוד זה מממש את הממשק באופן מפורש. הוא מסמן כל שיטה בשם הממשק:

```
void ITokenVisitor.VisitComment(string token)
{
   Write(token, Color.Black);
}
void ITokenVisitor.VisitIdentifier(string token)
{
   Write(token, Color.Black);
}
void ITokenVisitor.VisitKeyword(string token)
{
   Write(token, Color.Blue);
}
void ITokenVisitor.VisitOperator(string token)
{
   Write(token, Color.Black);
}
void ITokenVisitor.VisitPunctuator(string token)
{
   Write(token, Color.Black);
}
void ITokenVisitor.VisitStringLiteral(string token)
{
   Write(token, Color.Green);
}
void ITokenVisitor.VisitWhitespace(string token)
{
   Write(token, Color.Black);
}
```

טיפ

באפשרותך להקליד שיטות אלו לחלון Code and Text Editor באופן ישיר, או לחילופין, תוכל להשתמש בתכונה של Visual Studio 2005 המאפשרת להוסיף מימושי ברירת מחדל ולאחר מכן להוסיף את הקוד המתאים לגוף השיטה. כדי לעשות זאת, לחץ לחיצה ימנית על המזהה **ITokenVisitor** הנמצא במשפט הבא:

```
sealed class ColorSyntaxVisitor : ITokenVisitor
```

בתפריט הקיצור שיופיע, הצבע עם סמן העכבר על Implement Interface ולחץ על Implement Interface Explicitly (על ידי Shift + Alt + F10). כל אחת מהשיטות שתיווצרנה תכיל משפט הזורק חריג **Exception** שנושא את ההודעה "This method or operation is not implemented" (כלומר, שיטה או פעולה זו אינה מיושמת). החלף את הקוד הקיים בקוד שהוצג קודם לכן.

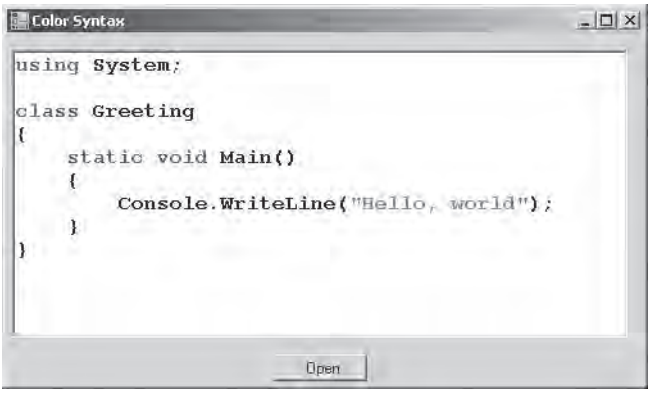
7. בתפריט Build בחר Build Solution. תקן שגיאות ובנה שוב במידת הצורך.

8. בתפריט Debug בחר Start Without Debugging.

כתוצאה יופיע הטופס Color Syntax.

9. לחץ על Open שבטופס.

הקוד המשוכפל יוצג בתיבת הטקסט העשיר, כאשר מילות המפתח מסומנות בכחול והליטראלים בירוק.

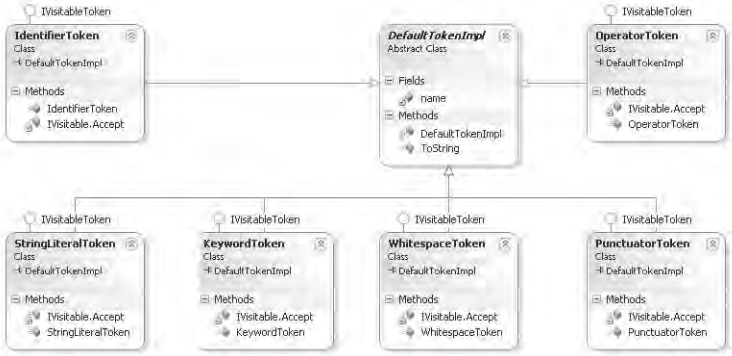


10. סגור את הטופס וחזור אל Visual Studio 2005.

יצירת דיאגרמה של מחלקה

החלון Class View הוא כלי שימושי להצגת ההיררכיה של המחלקות והממשקים בפרויקט. סביבת הפיתוח גם מאפשרת ליצור דיאגרמות אשר מציגות את אותו המידע בצורה ויזואלית (ניתן להשתמש בדיאגרמה גם כדי להוסיף מחלקות וממשקים חדשים וגם כדי להגדיר שיטות, מאפיינים וחברי מחלקה נוספים).

כדי ליצור דיאגרמה חדשה של מחלקה, פתח את תפריט Project ובחר Add New Item. בחלון Add New Item בחר בתבנית Class Diagram ולחץ Add. פעולה זו תחולל דיאגרמה ריקה, שתוכל להשתמש בה כדי ליצור סוגים חדשים על ידי כך שתגרור פריטים מהקטגוריה Class Designer שבערכת הכלים. כדי ליצור דיאגרמה מכל המחלקות הקיימות, גרור אותן זו אחרי זו מהחלון Class View, או לחילופין גרור רק את מרחב השמות שהן משתייכות אליו. הדיאגרמה מציגה את היחסים בין המחלקות והממשקים, ובאפשרותך להרחיב את ההגדרות של כל אחת מהמחלקות בכדי לצפות בתכולתן. ניתן לשנות את מיקום המחלקות והממשקים באמצעות גרירה, כדי להפוך את הדיאגרמה לברורה יותר, כפי שמוצג להלן:



סיכום שילובי מילות המפתח

הטבלה הבאה מסכמת את שילובי החוקיים (yes) והבלתי חוקיים (no) של מילות המפתח.

Keyword	Interface	Abstract class	Class	Sealed class	struct
Abstract	no	yes	no	no	no
New	yes *	yes	yes	yes	no †
override	no	yes	yes	yes	no ‡
private	no	yes	yes	yes	yes
protected	no	yes	yes	yes	no **
public	no	yes	yes	yes	yes
sealed	no	yes	yes	yes	no
virtual	no	yes	yes	no	no

* ממשק יכול להרחיב ממשק אחר ולהציג שיטה חדשה אשר נושאים בחתימה זהה לשלו.

† מבנה (struct) נגזר באופן מפורש מהמחלקה System.Object, אשר מכילה שיטות שהמבנה יכול להסתיר.

‡ מבנה נגזר באופן מפורש מהמחלקה System.Object, אשר אינה מכילה שיטות וירטואליות כלל.

** מבנה לעולם יהיה חתום, ולכן לא ניתן לגזור ממנו.

מחלקת הבסיס האולטימטיבית: Object

כל דבר ב-#C הוא מחלקה. המחלקה **Object** היא מחלקת הבסיס האולטימטיבית ב-#C. במילים אחרות, **Object** היא מחלקת השורש בהיררכיית המחלקות של NET framework.. המשמעות היא, שזו מחלקת הבסיס הראשונה.

על יסוד מנגנון ההורשה שלמדת בפרק זה, **כל דבר ב-#C הוא Object**. המשמעות היא שכל סוגי הנתונים והמחלקות האחרות **נגזרים** מהמחלקה **Object**, וכל השיטות הזמינות במחלקה **Object** זמינות לכל מחלקות NET., יש שתי שיטות מעניינות שנכללות במופעים של Object, ולכן כל המחלקות מכילות אותן, אלו הן השיטות **GetType** ו-**ToString**. השיטה **GetType** מחזירה את סוג הנתונים של האובייקט, והשיטה **ToString** מחזירה מחרוזת המייצגת את האובייקט הנוכחי.

☯ **אם ברצונך להמשיך לפרק הבא**

השאר את Visual Studio 2005 פעילה ועבור לפרק 13.

☯ **אם ברצונך לכבות כעת את Visual Studio 2005**

פתח את תפריט Menu ולחץ Exit. אם מופיעה תיבת דו-שיח Save, לחץ Yes.

פרק 12 – טבלה מסכמת

המשימה	צריך

ליצור מחלקה נגזרת ממחלקת בסיס.	להכריז על שם מחלקה חדש, לאחריו נקודתיים, ולבסוף שם מחלקת הבסיס. לדוגמה:
	`class Derived : Base`
	`{`
	`  ...`
	`}`

לקרוא לבנאי של מחלקת בסיס.	לספק רשימת פרמטרים עבור הבנאי לפני הגוף הבנאי של המחלקה הנגזרת. לדוגמה:
	`class Derived : Base`
	`{`
	`  ...`
	`  public Derived(int x) : Base(x)`
	`  {`
	`  ...`
	`  }`
	`  ...`
	`}`

להכריז על שיטה וירטואלית.	להשתמש במילת המפתח **virtual** בעת ההכרזה על השיטה. לדוגמה:
	`class Mammal`
	`{`
	`  public virtual void Breathe()`
	`  {`
	`  ...`
	`  }`
	`  ...`
	`}`

להכריז על ממשק.	להשתמש במילת המפתח **interface**. לדוגמה:
	`interface IDemo`
	`{`
	`  string Name();`
	`  string Description();`
	`}`

לממש ממשק.	להכריז על מחלקה שמשתמשת בתחביר זהה לזה של מחלקה נגזרת, ולאחר מכן לממש את כל הפונקציות של הממשק. לדוגמה:
	`class Test : IDemo`
	`{`
	`  public string IDemo.Name()`
	`  {`
	`  ...`
	`  }`
	`  public string IDemo.Description()`
	`  {`
	`  ...`
	`  }`
	`}`

פרק 13

איסוף אשפה וניהול משאבים

בסיום פרק זה, תוכל:

- לנהל משאבים של מערכת בעזרת **איסוף אשפה** (garbage collection).
- לכתוב קוד המופעל כאשר **מפרק** (destructor) מסיים את חיי האובייקט.
- לשחרר משאב בנקודת זמן ידועה מראש באופן המונע חריגים, באמצעות משפט **try/finally**.
- לשחרר משאב בנקודת זמן ידועה מראש באופן המונע חריגים, באמצעות משפט **using**.

בפרקים הקודמים למדת ליצור משתנים ואובייקטים, והבנת כיצד הזיכרון מוקצה לאחסון משתנים ואובייקטים בעת יצירתם. לשם תזכורת: סוגי ערך (value types) מאוחסנים במחסנית (stack), בעוד שסוגי הפניה (reference type) מאוחסנים במצבור (heap). כפי שידוע לך, למחשבים אין כמות בלתי מוגבלת של זיכרון, ולכן יש לשחרר את הזיכרון ברגע שאין עוד צורך במשתנה או אובייקט מסוים. סוגי ערך שונים מושמדים, והזיכרון שהיה מאוחסן בו מתפנה כאשר השיטה שבמסגרתה נוצר מסיימת את תפקידה. אבל זה החלק הפשוט. מה קורה לסוגי הפניה? אובייקטים נוצרים כתוצאה משימוש במילת המפתח **new**, אולם כיצד ומתי האובייקט מושמד? פרק זה עוסק בסוגיה זו.

אורך חיי האובייקט

הבה נראה מה קורה כאשר יוצרים ומשמידים אובייקט.

אובייקט יוצרים בדרך זו:

```
TextBox message = new TextBox(); // TextBox is a reference type
```

לכאורה נראה שהתהליך **new** הוא חד-שלבי, אולם למעשה תהליך יצירת אובייקט הוא תהליך דו-שלבי. ראשית, יש להקצות זיכרון גולמי מהמצבור. לך כמתכנת, אין שליטה על שלב זה ביצירת האובייקט. שנית, יש להמיר את הזיכרון הגולמי לאובייקט. כלומר, לאתחל את האובייקט. על תהליך זה אתה יכול לשלוט באמצעות בנאי.

הערה

מתכנתים מרקע של ++C צריכים לשים לב שבשפת #C לא ניתן להעמיס את **new** כדי לשלוט בהקצאת הזיכרון.

לאחר יצירת אובייקט, ניתן לגשת אל חברי האובייקט באמצעות האופרטור נקודה (.). לדוגמה:

```
message.Text = “People of Earth, your attention please”;
```

אתה יכול לגרום למשתני הפניה אחרים להפנות אל אותו אובייקט:

```
TextBox ref = message;
```

מספר ההפניות שניתן ליצור לאובייקט אחד בלתי מוגבל, אך סביבת ההרצה (runtime) צריכה לנהל מעקב על כל ההפניות הללו. גם אם המשתנה **message** ייעלם (על ידי כך שייצא מתחום ההכרזה), משתנים אחרים (כמו **ref**, לדוגמה) יישארו פעילים. לפיכך, אורך החיים של אובייקט מסוים אינו יכול להיקבע על פי אורך חיי משתנה הפניה בודד. ניתן לחסל אובייקט רק לאחר שנעלמות כל ההפניות אליו.

הערה

מתכנתים מרקע של ++C צריכים לשים לב שבשפת #C אין אופרטור **delete**. השליטה בחיסול אובייקט נמצאת בידי סביבת ההרצה.

בדומה ליצירת אובייקטים, גם חיסול אובייקטים הוא תהליך דו-שלבי. שני שלבי ההריסה הינם נגדיים לשני שלבי היצירה. ראשית, עליך לכתוב **מפרק** (destructor). שנית, יש להחזיר את הזיכרון הגולמי למצבור; זהו קטע הזיכרון אשר שימש לאחסון האובייקט וחייב להיות זמין שוב לשימוש. אין לך כל שליטה על שלב זה בתהליך. תהליך פירוק האובייקט והחזרת הזיכרון למצבור נקרא **איסוף אשפה** (**garbage collection**).

כתיבת מפרקים

מפרק (destructor) משמש לביצוע כל הפעולות הנחוצות בעת איסוף אשפה. התחביר ליצירת מפרק הוא סימן טילדה (~) ולאחריו שם המחלקה. לפניך דוגמה למחלקה פשוטה אשר סופרת את מספר המופעים הפעילים באמצעות הגדלה באחד של מונה סטטי בגוף הבנאי, ומקטינה באחד את המונה הסטטי בגוף המפרק:

```
class Tally
{
   public Tally()
   {
      instanceCount++;
   }
   ~Tally()
   {
```

```
      instanceCount--;
   }
   public static int InstanceCount()
   {
      return instanceCount;
   }
   ...
   private static int instanceCount = 0;
}
```

יש מספר הגבלות חשובות החלות על המפרק:

- לא ניתן להכריז על מפרק במבנה (struct). מבנה הוא סוג ערך המאוחסן במחסנית ולא במצבור, ולכן איסוף האשפה אינו תקף לגביו. הנה דוגמה למצב שגיאה:

```
struct Tally
{
   ~Tally() { ... } // compile-time error
}
```

- לא ניתן להכריז על רמת הנגישות (כגון public) של המפרק. הסיבה לכך היא שלעולם לא תקרא למפרק בעצמך, כי אוסף האשפה יקרא לו.

```
public ~Tally() { ... } // compile-time error
```

- לא ניתן להכריז על מפרק עם פרמטרים, והמפרק אינו מסוגל לקבל פרמטרים. גם הפעם הסיבה לכך היא שאינך יכול לקרוא למפרק בעצמך.

```
~Tally(int parameter) { ... } // compile-time error
```

- כל מפרק מומר באופן אוטומטי על ידי המהדר לדריסה, או עקיפה (override) של השיטה **Object.Finalize**. כאשר המהדר פוגש את המפרק הזה:

```
class Tally
{
   ~Tally() { ... }
}
```

הוא ממיר אותו לקוד הזה:

```
class Tally
{
   protected override void Finalize()
   {
      try { ... }
      finally { base.Finalize(); }
   }
}
```

השיטה **Finalize** אשר המהדר יוצר מכילה את גוף המפרק בתוך בלוק **try**, כשאחריו בלוק **finally** אשר קורא למחלקת הבסיס **Finalize** (ראה פרק 6 למידע על מילות המפתח **try** ו-**finally**). הדבר מבטיח שהמפרק תמיד יקרא בסופו של דבר למפרק של מחלקת הבסיס. חשוב להבין שרק המהדר יכול לבצע את ההמרה הזאת. אינך יכול לעקוף את **Finalize** בעצמך, ואינך יכול לקרוא ל-**Finalize** בעצמך.

מדוע צריך אוסף אשפה?

בשפת #C אינך יכול לחסל אובייקטים בעצמך. אין תחביר מתאים לעשות זאת, ולאלה שפיתחו את #C הייתה סיבה טובה לא ליצור תחביר כזה. אם האחריות לחיסול האובייקטים הייתה בידך, במוקדם או במאוחר היה מתממש אחד מהתרחישים הבאים:

- תשכח לחסל אובייקט. המשמעות היא שהמפרק של האובייקט (אם בכלל יש לו מפרק) לא יופעל, והזיכרון ששימש לאחסון האובייקט לא יוחזר למצבור. במצב כזה הזיכרון במחשב שלך עשוי להיות עמוס באובייקטים שאינך זקוק להם עוד.
- תנסה לחסל אובייקט פעיל. זכור, הגישה לאובייקטים מתבצעת באמצעות הפניות. אם במחלקה קיימת הפניה לאובייקט שחוסל, ההפניה תהיה **הפניה תלויה** (dangling reference). ההפניה התלויה עשויה להפנות לזיכרון שאינו בשימוש או אפילו לאובייקט אחר שנמצא באותו מקום בזיכרון. בכל מקרה, תוצאות השימוש בהפניה תלויה לא יהיו צפויים מראש, ועלולות לפגוע בתוכנית.
- תנסה לחסל את אותו אובייקט מספר פעמים. פעולה זו יכולה להסתיים מבלי לפגוע בתוכנית, אולם היא עשויה להיות הרסנית. הדבר תלוי בקוד שמופיע במפרק.

בעיות אלו אינן מתקבלות על הדעת בשפת #C, אשר יעדי העיצוב העיקריים שלה הם יציבות ובטיחות. לכן, אוסף האשפה ממלא עבורך את תפקיד חיסול האובייקטים, ומוודא את קיום התנאים הבאים:

- במוקדם או במאוחר כל אובייקט יחוסל והמפרקים שלו יופעלו. בסיום התוכנית, יחוסלו כל האובייקטים הנותרים.
- כל אובייקט יחוסל פעם אחת בלבד.
- אובייקט יחוסל רק לאחר שיהפוך לבלתי נגיש. כלומר, כאשר לא יהיו יותר הפניות לאותו אובייקט.

הבטחות אלו שימושיות ביותר ופוטרות אותך, המתכנת, מהצורך לנקות אחריך, פעולה שהיא מורכבת הרבה יותר ממה שהיא נשמעת. כך תוכל להתרכז בכתיבת הלוגיקה של התוכנית עצמה ולהפיק יותר מעבודתך.

מתי מתרחש איסוף האשפה? שאלה זו מוזרה במקצת, כי למעשה פינוי האשפה מתבצע כאשר אין עוד צורך באובייקט. זה אמנם נכון, אולם הפעולה אינה תמיד מיידית. איסוף אשפה עשוי להיות תהליך מורכב, ולכן סביבת ההרצה אוספת אשפה רק בעת הצורך (כאשר הזיכרון הפנוי אוזל ואינו מאפשר כניסת אובייקטים חדשים), ואז האיסוף מחסל כמה שיותר אשפה ומפנה זיכרון רב יותר. ביצוע מספר קטן של איסופים גדולים יעיל הרבה יותר מאיסוף נקודתי.

הערה

ניתן להפעיל את איסוף האשפה בתוכנית על ידי קריאה לשיטה הסטטית **System.GC.Collect()**. אך מלבד מקרים נדירים, אין זה מומלץ.

השיטה **System.GC.Collect()** מפעילה את אוסף האשפה, אולם התהליך פועל בלוח זמנים משלו, וגם לאחר סיום הקריאה לשיטה, לא ניתן לדעת אם האובייקטים הושמדו כבר או לא. לכן, כדאי מוטב להשאיר לסביבת ההרצה את ההחלטה מתי לנקות את הזיכרון.

מאפיין נוסף של אוסף האשפה הוא שאינך יודע את סדר החיסול של האובייקטים, ולכן גם אינך יכול להסתמך על כך. הנקודה האחרונה, ואולי החשובה ביותר, שחשוב להבין היא, שהמפרקים אינם מופעלים עד הרגע שבו האובייקטים נמחקים. אם אתה כותב מפרק אתה יכול לדעת שהוא יופעל במועד כלשהו, אולם אינך יכול לדעת מתי.

כיצד פועל אוסף האשפה?

אוסף האשפה פועל באופן עצמאי ויכול לפעול בזמנים מסוימים בלבד (בעיקר כאשר שיטה ביישום מסתיימת). כאשר הוא פועל, פעילויות אחרות במסגרת התוכנית שלך מופסקות באופן זמני. הסיבה לכך היא שאוסף האשפה עשוי להעביר אובייקטים ממקום למקום ולעדכן את ההפניות, והוא אינו יכול לעשות זאת כאשר יש אובייקטים בשימוש. אלה הפעולות של אוסף האשפה:

1. בונה מפה של כל האובייקטים הפעילים. הוא עושה זו באמצעות מעקב אחר השדות שמפנים אשפה בתוך האובייקטים. אוסף האשפה בונה את המפה באופן זהיר ביותר ומוודא שההפניות המחזוריות אינן יוצרות לולאות אין סופיות. כל אובייקט **שאינו** מופיע במפה זו נחשב לבלתי-נגיש.
2. בודק אם יש מבין האובייקטים הבלתי נגישים כאלה שיש להם מפרק (destructor) שצריך להפעילו (תהליך שנקרא **סיום**, **finalization**). אובייקטים בלתי נגישים אשר צריכים לעבור סיום מצורפים לתור מיוחד בשם **freachable queue** (יש לבטא F-reachable).
3. הקצאת הזיכרון של שאר האובייקטים (שאין צורך לסיים אותם) נלקחת מהם באמצעות דחיפת האובייקטים הנגישים במורד הרשימה. פעולה זו גורמת לאיחוי המצבור ושחרור הזיכרון שבמעלה המצבור. כאשר אוסף האשפה מזיז אובייקט, הוא גם מעדכן את ההפניות אליו.
4. שאר הפעולות בתוכנית יכולות להמשיך כרגיל.
5. אוסף הזבל מסיים (finalize) בפעולה נפרדת את האובייקטים הבלתי נגישים שיש לסיים (ואשר נמצאים בתור freachable).

המלצות

כתיבת מחלקות המכילות מפרקים מוסיפה למורכבות הקוד ולתהליכי איסוף האשפה, וגם גורמת לתוכנית לפעול יותר לאט. אם אין מפרקים בתוכנית שלך, אוסף האשפה אינו צריך לבצע את פעולות 3 ו-5 שבסעיף הקודם. אין ספק שאי-ביצוע פעולה חוסך זמן לעומת הביצוע שלה. לפיכך, נסה להשתמש במפרקים רק במקרים שחיוניים לפעולת התוכנית. פתרון טוב יהיה להשתמש במשפט **using** כתחליף למפרקים (על משפטי **using** נדבר בהמשך הפרק).

נקוט משנה זהירות בעת כתיבת מפרק. חשוב במיוחד להיות מודע שבמקרה שהמפרק שלך קורא לאובייקטים אחרים, ייתכן מאוד שהמפרקים שלהם כבר הופעלו על ידי אוסף האשפה. זכור שסדר סיום הפעילות של האובייקטים אינו קבוע, ולכן עליך לוודא שהמפרקים שלך לא תלויים זה בזה, או חופפים זה עם זה (אל תבנה שני מפרקים אשר מנסים לשחרר את אותו המשאב, לדוגמה).

ניהול משאבים

לעיתים כדאי לשחרר משאב באמצעות מפרק. יש משאבים חשובים מדי מכדי לתת להם להישאר תפוסים שלא לצורך למשך זמן שרירותי. יש לשחרר משאבים כאלה, ולעשות זאת מוקדם ככל הניתן. במצבים אלה, הברירה היחידה הניצבת בפניך היא לשחרר את המשאב בעצמך. שיטת **הפינוי** (**disposal**) מיועדת לפינוי משאבים. אם למחלקה יש שיטת פינוי, באפשרותך לקרוא לה באופן מפורש וכך לשלוט בתזמון של פינוי המשאב.

שיטות פינוי

דוגמה למחלקה אשר מממשת שיטת פינוי היא **TextReader** ממרחב השמות **System.IO**. המחלקה מכילה מנגנון לקריאת תווים מתוך זרם (stream) שנקלט בה. המחלקה **TextReader** כוללת שיטה וירטואלית **Close**, אשר סוגרת את הזרם. המחלקה **StreamReader** (הקוראת תווים מתוך זרם, כגון קובץ פתוח) והמחלקה **StringReader** (הקוראת תווים מתוך מחרוזת) נגזרות שתיהן מ-**TextReader**, ושתיהן עוקפות את השיטה **Close**. בדוגמה הבאה מתבצעת קריאה של שורות טקסט מתוך קובץ באמצעות המחלקה **StreamReader**, ולאחר מכן הן מוצגות על המסך:

```
TextReader reader = new StreamReader(filename);
string line;
while ((line = reader.ReadLine()) != null)
{
   Console.WriteLine(line);
}
reader.Close();
```

השיטה **ReadLine** קוראת את שורת הטקסט הבאה מהקובץ לתוך מחרוזת. השיטה **ReadLine** מחזירה ערך null אם לא נותר דבר כלשהו בשטף. חשוב לקרוא לשיטה **Close** בסיום העבודה עם **reader** כדי לשחרר את מטפל הקובץ (file handle) ואת המשאבים הרלוונטיים. דוגמה זו בעייתית, מכיוון שאינה חסינה בפני חריגים. אם הקריאה לשיטה **ReadLine** (או **WriteLine**) זורקת חריג, הקריאה לשיטה **Close** לא תתבצע. אם הדבר יקרה יותר מדי פעמים, בסופו של דבר לא יישארו יותר מטפלי קובץ, ולא ניתן יהיה לפתוח קבצים נוספים.

פינוי חסין בפני חריגים

אחת הדרכים לוודא ששיטת פינוי (כגון Close) תמיד תופעל, גם במקרה של חריג, היא לקרוא לשיטה במסגרת בלוק finally. הנה שוב הדוגמה הקודמת, אולם הפעם עם שימוש בטכניקה האמורה:

```
TextReader reader = new StreamReader(filename);
try
{
   string line;
   while ((line = reader.ReadLine()) != null)
   {
      Console.WriteLine(line);
```

```
    }
}
finally
{
    reader.Close();
}
```

שימוש בבלוק **finally** אכן פותר את הבעיה, אולם פתרון זה אינו אידיאלי מכיוון שיש לו מספר חסרונות:

- כאשר צריך לפנות מספר רב של משאבים, הקוד יהיה מסובך (מספר רב של בלוקים מסוג **try** ו-**finally** זה בתוך זה).
- במקרים מסוימים ייתכן שיהיה צורך לשנות את הקוד (לדוגמה, לשנות את מבנה ההכרזה של ההפניה למשאב, לאתחל את ההפניה לערך **null**, ולוודא שההפניה אינה **null** בבלוק finally).
- פתרון זה יוצר פתרון מסורבל. כלומר, הפתרון קשה להבנה וצריך לחזור על הקוד בכל פעם שצריך לבצע פעולה מסוג זה.
- ההפניה למשאב נשארת בתחום ההכרזה (scope) גם לאחר הבלוק **finally**. משמעות הדבר היא שייתכן שתנסה בטעות להשתמש במשאב לאחר שכבר פינו אותו.

המשפט **using** נועד לפתור את כל הבעיות הללו.

משפט using

המשפט **using** מאפשר לשלוט באורך החיים של המשאבים בצורה אלגנטית. תוכל ליצור אובייקט ולהשתמש בו, והוא יחוסל כאשר הבלוק של המשפט **using** יגיע לסופו.

חשוב

אל תתבלבל בין המשפט using שבפרק זה לבין ההנחיה **using**, שתפקידה לצרף תחומי שמות אל תחום ההכרזה. למרבה הצער, למילת המפתח הזו יש שתי משמעויות שונות.

להלן התחביר עבור המשפט **using**:

```
using ( type variable = initialization ) {embeddedStatement}
```

זו הדרך הטובה ביותר לוודא שהקוד שלך תמיד יפעיל את השיטה **Close** על **TextReader**:

```
using (TextReader reader = new StreamReader(filename))
{
    string line;
    while ((line = reader.ReadLine()) != null)
    {
        Console.WriteLine(line);
    }
}
```

משפט **using** שהוצג לעיל פועל באופן זהה לחלוטין לקוד הבא:

```
{
   TextReader reader = new StreamReader(filename);
   try
   {
      string line;
      while ((line = reader.ReadLine()) != null)
      {
         Console.WriteLine(line);
      }
      }
   finally
   {
      if (reader != null)
      {
         ((IDisposable)reader).Dispose();
      }
   }
}
```

הערה

שים לב לתחום ההכרזה של הבלוק החיצוני. הסידור הזה גורם לכך שמשתנה אשר יוגדר בתחומי המשפט **using** יצא מתחום ההכרזה בסוף הבלוק.

המשתנה שאתה מכריז עליו במסגרת המשפט **using** חייב להיות מסוג אשר יכול לממש את הממשק **IDisposable**. הממשק **IDisposable** נמצא במרחב השמות **System** ומכיל שיטה אחת בלבד אשר נקראת **Dispose**:

```
namespace System
{
   interface IDisposable
   {
      void Dispose();
   }
}
```

המחלקה **StreamReader** מיישמת את הממשק **IDisposable**, והשיטה **Dispose** קוראת לשיטה **Close** כדי לסגור את זרם הנתונים (stream). משפט **using** מהווה פתרון אלגנטי, עמיד בפני חריגים ויציב, אשר מבטיח שהמשאבים יפונו תמיד באופן אוטומטי. כך נפתרות כל הבעיות שהתעוררו בפתרון המשתמש ב-**try/finally** באופן ידני. הפתרון הנוכחי:

- מתאים בגודלו כאשר שצריך לפנות משאבים אחדים.
- אינו מתנגש עם הלוגיקה של קוד התוכנית.
- פותר את הבעיה באופן מופשט ומונע חזרות מיותרות.
- הפתרון יציב. לא ניתן להשתמש במשתנים שהוכרזו במסגרת המשפט using (במקרה זה, **reader**) לאחר סיום המשפט, מכיוון שהם לא יהיו יותר בתחום ההכרזה. כל ניסיון לעשות זאת יגרום לשגיאה בשלב ההידור.

קריאה לשיטה Dispose מתוך המפרק

בעת כתיבת מחלקה, האם עליך לכתוב מפרק (destructor) או לממש את הממשק **IDisposable**? קריאה למפרק תתבצע בכל מקרה, אלא שלא ניתן לדעת מתי. עם זאת, ניתן לדעת בדיוק מתי תתבצע הקריאה לשיטה **Dispose**. אינך יכול להיות בטוח אם הקריאה תתבצע, מכיוון שהדבר מותנה בכך שהמתכנת ישתמש במשפט **using**. כדי להבטיח שהשיטה **Dispose** תופעל תמיד, עליך לקרוא לה מתוך המפרק. פעולה זו מהווה גיבוי יעיל ונוח. גם במקרים שבהם תשכח לקרוא לשיטה **Dispose**, תוכל לפחות להיות בטוח שהיא תופעל במוקדם או במאוחר, גם אם זה יקרה רק לאחר סיום פעולת התוכנית. ניתן לעשות זאת באופן הבא:

```
class Example : IDisposable
{
    ...
    ~Example()
    {
        Dispose();
    }

    public virtual void Dispose()
    {
        if (!this.disposed)
        {
            try
            {
                // release scarce resource here
            }
            finally
            {
                this.disposed = true;
                GC.SuppressFinalize(this);
            }
        }
    }
    public void SomeBehavior() // example method
    {
        checkIfDisposed();
        ...
    }
    ...
    private void checkIfDisposed()
    {
        if (this.disposed)
        {
            throw new ObjectDisposedException("Example");
        }
    }
    private Resource scarce;
    private bool disposed = false;
}
```

שים לב לנקודות הבאות:

- המחלקה מממשת את **IDisposable**.
- המפרק קורא ל-**Dispose**.
- השיטה **Dispose** היא ציבורית וניתן לקרוא לה בכל עת.
- ניתן לקרוא לשיטה **Dispose** כמה וכמה פעמים מבלי לפגוע בתוכנית. המשתנה **disposed** מציין אם השיטה כבר הופעלה. המשאב היקר משוחרר רק בהפעלה הראשונה של השיטה.
- השיטה **Dispose** קוראת לשיטה הסטטית **GC.SuppressFinalize**. שיטה זו מונעת את הפעלת המפרק על האובייקט על ידי אוסף האשפה.
- כל השיטות הרגילות במחלקה (כגון **SomeBehavior**) בודקות האם האובייקט כבר פונה או לא. אם כן, הן זורקות חריג.

אבטחת הקוד בפני חריגים

בתרגיל הבא עליך לכתוב קוד קצר אשר יגן על התוכנית בפני **חריגים** (exceptions). התוכנית פותחת קובץ טקסט, קוראת את תכולתו שורה אחר שורה, כותבת את השורות הללו בתיבת טקסט עשיר (rich text box) שעל טופס Windows ולבסוף סוגרת את קובץ הטקסט. אולם, אם מתעורר חריג בעת קריאת הקוד או בעת כתיבת השורות בתיבת הטקסט, הקריאה לסגירת הקובץ לא תתבצע. עליך לשכתב את הקוד באמצעות משפט **using**, וכך להבטיח שהקוד יהיה עמיד בפני חריגים.

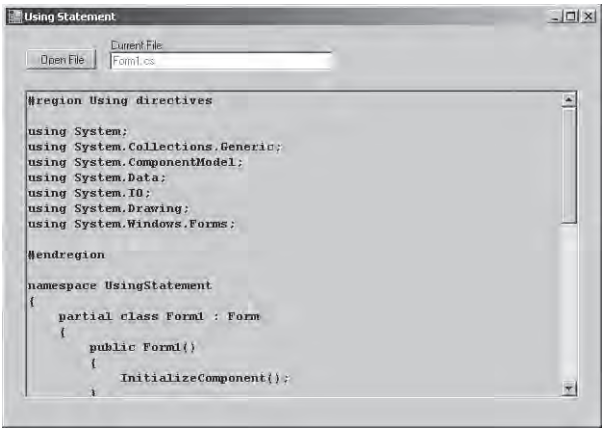
כתוב משפט using

1. הפעל את סביבת הפיתוח Microsoft Visual Studio 2005.
2. פתח את הפרויקט **UsingStatment** שבתיקייה My Documents\Microsoft Press\Visual CSharp Step by Step\Chapter 13\UsingStatement.
3. בתפריט Debug בחר Start Without Debugging.

   הטופס Windows יופיע על המסך.
4. לחץ על Open שבטופס.
5. בתיבת הדו-שיח Open, נווט אל התיקייה My Documents\Microsoft Press\Visual CSharp Step by Step\Chapter 13\UsingStatement.

   בחר כעת את קובץ המקור Form1.cs, שהינו קובץ המקור של היישום עצמו.
6. לחץ Open.

   תכולת הקובץ תופיע על טופס Windows.



7. סגור את הטופס וחזור לסביבת הפיתוח.

8. הצג את קובץ המקור Form1.cs בחלון Code and Text Editor ואתר את השיטה **openFileDialog_FileOk**.

```
private void openFileDialog_FileOk(object sender,System.
    ComponentModel.CancelEventArgs e)
{
    string fullPathname = openFileDialog.FileName;
    FileInfo src = new FileInfo(fullPathname);
    filename.Text = src.Name;
    source.Text = “”;
    TextReader reader = new StreamReader(fullPathname);
    string line;
    while ((line = reader.ReadLine()) != null)
    {
        source.Text += line + “\n”;
    }
    reader.Close();
}
```

שלושת המשתנים **filename**, **openFileDialog** ו-**source** הם שדות פרטיים במחלקה **Form1**. הבעיה עם הקוד היא, שהקריאה לשיטה **reader.Close** אינה ודאית. אם ייזרק חריג לאחר פתיחת הקובץ, הוא יגרום לשיטה להסתיים, אולם הקובץ יישאר פתוח עד שהיישום ייפסק.

9. שכתב את השיטה openFileDialog_FileOk באמצעות משפט using בדרך זו:

```
private void openFileDialog_FileOk(object sender, System.
ComponentModel.CancelEventArgs e)
{
    string fullPathname = openFileDialog.FileName;
    FileInfo src = new FileInfo(fullPathname);
    filename.Text = src.Name;
    source.Text = “”;
```

```
    using (TextReader reader = new StreamReader(fullPathname))
    {
        string line;
        while ((line = reader.ReadLine()) != null)
        {
            source.Text += line + "\n";
        }
    }
}
```

שים לב שאין יותר צורך לקרוא ל- **reader.Close** מכיוון שהוא יופעל על ידי השיטה **Dispose** מהמחלקה **StreamReader** בסיום המשפט **using**. אין זה משנה אם המשפט using יסתיים באופן טבעי או אם הוא ייקטע על ידי חריג, כי הקובץ ייסגר בכל אחד מהמקרים.

10. בנה מחדש והפעל שוב את היישום כדי לוודא שהכל עדיין פועל כשורה.

☯ **אם ברצונך להמשיך לפרק הבא**

השאר את Visual Studio 2005 פעילה ועבור לפרק 14.

☯ **אם ברצונך לכבות כעת את Visual Studio 2005**

פתח את תפריט Menu ולחץ Exit. אם מופיעה תיבת דו-שיח Save, לחץ Yes.

פרק 13 – טבלה מסכמת

המשימה	צריך

לכתוב מפרק (destructor).	לכתוב שיטה אשר שמה זהה לשם המחלקה ואשר לפניה מופיע התו טילדה (~). לא ניתן לשנות את הגישה למחלקה זו (פרטית/ציבורית) והיא אינה יכולה לקבל פרמטרים או להחזיר ערך. לדוגמה:
	`class Example`
	`{`
	`  ~Example()`
	`  {`
	`  ...`
	`  }`
	`}`

לקרוא למפרק.	לא ניתן לקרוא למפרק. רק אוסף האשפה יכול לקרוא לו.
לאלץ איסוף אשפה.	לקרוא לשיטה **System.GC.Collect**.

לשחרר משאב בנקודת זמן ידועה מראש.	לכתוב שיטת פינוי (אשר מפנה משאבים – disposal method), ולקרוא לה מהתוכנית באופן מפורש. לדוגמה:
	`class TextReader`
	`{`
	`  ...`
	`  public virtual void Close()`
	`  {`
	`  ...`
	`  }`
	`}`
	`class Example`
	`{`
	`  void Use()`
	`  {`
	`  TextReader reader = ...;`
	`  // use reader`
	`  reader.Close();`
	`  }`
	`}`

המשימה	צריך
לשחרר משאב בנקודת זמן ידועה מראש, ובצורה עמידה בפני חריגים.	לשחרר את המשאב באמצעות משפט using. לדוגמה:

```
class TextReader : IDisposable
{
    ...
    public virtual void Dispose()
    {
        // calls Close
    }
    public virtual void Close()
    {
        ...
    }
}
class Example
{
    void Use()
    {
        using (TextReader reader = ...)
        {
            // use reader
        }
    }
}
```

חלק ג'

יצירת רכיבים

בחלק זה:

פרק 14

מימוש מאפיינים עבור גישה לתכונות

כאשר תסיים פרק זה, תוכל:

- לכמס (encapsulate) שדות לוגיים באמצעות מאפיינים (properties).
- לשלוט בגישת קריאה באמצעות אחראי גישה **get**.
- לשלוט בגישת כתיבה באמצעות אחראי גישה **set**.
- ליצור ממשק אשר מכריז על מאפיינים.
- לממש ממשק המכיל מאפיינים, באמצעות מבנים ומחלקות.

שני החלקים הראשונים של ספר זה עסקו בתחביר הבסיסי של שפת #C, ולימדו אותך כיצד להשתמש בשפה כדי לבנות סוגים חדשים של קוד, כגון מבנים (structs), רשימות (enums) ומחלקות (classes). למדת גם כיצד סביבת ההרצה מנהלת את הזיכרון המשמש את המשתנים והאובייקטים השונים בעת הפעלת תוכנית. כעת אתה כבר אמור להבין את מחזור החיים של אובייקטים שעליהם מבוססת למעשה שפת #C. הפרקים בחלק ג', העוסק ביצירת רכיבים, מתבססים על מה שלמדת, ומראים לך כיצד להשתמש ב-#C כדי ליצור רכיבים רב-שימושיים, ומחלקות מתפקדות, שניתן להשתמש בהן שוב ושוב ביישומים שונים. כלומר, לעשות שימוש חוזר בקוד שנכתב עבור פרויקט אחד גם עבור פרויקטים אחרים.

פרק זה עוסק בהגדרה של **מאפיינים** (properties) ובשימוש בהם להסתרת שדות במחלקה. בפרקים הקודמים הדגשנו שעליך להגדיר את השדות במחלקה כפרטיים, וליצור שיטות שייעודן לאחסן ערכים או לאחזר מהן ערכים. צורת עבודה זו אמנם מאפשרת גישה בטוחה ומסודרת, אולם התחביר המשמש לגישה לשדה בצורה כזו אינו טבעי. כאשר אתה רוצה לקרוא משתנה או לכתוב בו ערך, תשתמש בדרך כלל במשפט הצבה, ולכן ייתכן שקריאה לשיטה כדי לקרוא ולכתוב בשדה (אשר הוא למעשה סוג של משתנה) היא קצת מסורבלת. מטרת המאפיינים להתגבר על הבעייתיות הזו.

השוואה בין שדות לשיטות

תחילה עלינו להיזכר מדוע רצינו מראש להשתמש בשיטות כדי להסתיר שדות.

המבנה (**struct**) הבא מייצג נקודה על המסך באמצעות צמד קואורדינטות (**X**, **Y**):

```
struct ScreenPosition
{
    public ScreenPosition(int x, int y)
    {
        this.X = rangeCheckedX(x);
```

```
        this.Y = rangeCheckedY(y);
    }
    public int X;
    public int Y;
    private static int rangeCheckedX(int x)
    {
        if (x < 0 || x > 1280)
        {
            throw new ArgumentOutOfRangeException("X");
        }
        return x;
    }
    private static int rangeCheckedY(int y)
    {
        if (y < 0 || y > 1024)
        {
            throw new ArgumentOutOfRangeException("Y");
        }
        return y;
    }
}
```

הבעייתיות של מבנה זה היא שהוא אינו מקיים את כלל הזהב של הכימוס. המידע אינו פרטי, וכפי שלמדנו, מידע ציבורי הינו פתח לפורענות, מכיוון שלא ניתן לבדוק אותו ולשלוט בדרך הגישה אליו. לדוגמה, הבנאי **ScreenPosition** בודק שהפרמטרים שלו אינם חורגים מהטווח, אולם לא ניתן לבצע בדיקה שכזו עבור גישה ישירה לשדות ציבוריים. במוקדם או במאוחר (ולרוב, במוקדם), **X** או **Y** יקבלו ערך שאינו בטווח, כמו למשל כתוצאה מטעות של המתכנת:

```
ScreenPosition origin = new ScreenPosition(0, 0);
...
int xpos = origin.X;
origin.Y = -100; // Oops
```

הדרך הטובה ביותר לפתור את הבעיה היא להפוך את השדות לפרטיים וליצור **שיטת גישה** (accessor method) **ושיטת עריכה** (modifier method) כדי לקרוא ולכתוב בהתאמה את ערכי השדות הפרטיים. על שיטת העריכה תוטל משימת בדיקת הטווח של הערכים המוצבים בשדה, מכיוון שהבנאי בדק כבר את הערכים ההתחלתיים שלו. לדוגמה, הנה שיטת גישה (**GetX**) ושיטת עריכה (**SetX**) עבור השדה **X**. שים לב לבדיקה שמבצעת השיטה **SetX** לערכי הפרמטר:

```
struct ScreenPosition
{
    ...
    public int GetX()
    {
        return this.x;
    }
    public void SetX(int newX)
    {
        this.x = rangeCheckedX(newX);
    }
```

```
    ...
    private static int rangeCheckedX(int x) { ... }
    private static int rangeCheckedY(int y) { ... }
    private int x, y;
}
```

כעת הקוד מכיל בקרה מוצלחת של הטווחים התקניים. אולם לבקרת הטווח יש מחיר – התחביר של המבנה **ScreenPosition** כבר אינו תחביר טבעי לשדה, מכיוון שהוא מסורבל ומתבסס על שיטות. הדוגמה הבאה מגדילה את הערך של **X** ב-10. כדי לעשות זאת, יש לקרוא תחילה את הערך של **X** באמצעות שיטת הגישה **GetX**, ולאחר מכן לכתוב את הערך החדש של **X** על ידי שיטת העריכה **SetX**:

```
int xpos = origin.GetX();
origin.SetX(xpos + 10);
```

אם השדה **X** היה ציבורי, היינו יכולים להשתמש בקוד קצר הרבה יותר. הנה כך:

```
origin.X += 10;
```

אין ספק שבדרך תכנות זו, השימוש בשדות נקי, קצר ופשוט יותר. למרבה הצער, שימוש בשדות ציבוריים פוגע בכימוס. המאפיינים מאפשרים לשלב את היתרונות של שתי הדוגמאות: שימור הכימוס לצד תחביר דומה לזה של השדות.

מהם מאפיינים?

מאפיין (**property**) הוא הכלאה בין שדה לוגי לבין שיטה פיסית. השימוש במאפיינים זהה לחלוטין לשימוש בשדות, ומבחינה לוגית מאפיין נראה כמו שדה. המהדר מתרגם באופן אוטומטי את התחביר, הזהה לתחביר של שדות, לאחראי גישה (accessors) מיוחדים, הדומים לשיטות. הכרזה על מאפיין נראית כך:

```
AccessModifier Type PropertyName
{
    get
    {
        // read accessor code
    }
    set
    {
        // write accessor code
    }
}
```

מאפיין יכול להכיל שני בלוקים של קוד, המתחילים במילות המפתח **get** ו-**set**. הבלוק **get** מכיל משפטים המופעלים בעת קריאה מהמאפיין, והבלוק **set** מכיל משפטים המופעלים בעת כתיבה במאפיין. סוג המאפיין מקביל לסוג הנתונים הנכתבים במאפיין ונקראים ממנו באמצעות האחראים לגישה **set** ו-**get**.

הקוד הבא מציג את המבנה **ScreenPosition** לאחר ששוכתב באמצעות מאפיינים. בעת העיון בקוד עליך לשים לב לנקודות הבאות:

- **x** ו-**y**, באותיות רגילות, הם שדות **פרטיים**.
- **X** ו-**Y**, באותיות רישיות, הם מאפיינים **ציבוריים**.
- כל אחראי גישה מסוג **set** מקבל את הנתונים שעליו להציב דרך פרמטר מוסתר בשם **value**.

טיפ

שמות השדות והמאפיינים נקבעים על פי המוסכמות להקצאת שמות **פרטיים** ו**ציבוריים** של Microsoft Visual C#. שמות השדות והמאפיינים הציבוריים צריכים להתחיל באות רישית, בשעה ששמות השדות והמאפיינים הפרטיים צריכים להתחיל באות רגילה.

```
struct ScreenPosition
{
    public ScreenPosition(int X, int Y)
    {
        this.x = rangeCheckedX(X);
        this.y = rangeCheckedY(Y);
    }
    public int X
    {
        get { return this.x; }
        set { this.x = rangeCheckedX(value); }
    }
    public int Y
    {
        get { return this.y; }
        set { this.y = rangeCheckedY(value); }
    }
    private static int rangeCheckedX(int x) { ... }
    private static int rangeCheckedY(int y) { ... }
    private int x, y;
}
```

בדוגמה זו, לכל מאפיין יש מימוש ישיר על ידי שדה פרטי. יש דרכים נוספות למימוש מאפיין. הדרישה היחידה היא שאחראי הגישה **get** יחזיר ערך מהסוג המתאים. את הערך המוחזר ניתן לחשב בקלות, ובמקרה שכזה אין צורך בשדה פיסי.

הערה

למרות שהדוגמאות בפרק זה עוסקות בהגדרת מאפיינים עבור מבנה, אין קושי לעשות זאת גם עבור מחלקות. התחביר זהה לחלוטין.

שימוש במאפיינים

כאשר מאפיין הינו חלק מביטוי, הוא משמש בהקשר של קריאה (כאשר ערכו לא משתנה) או בהקשר של כתיבה (כאשר ערכו משתנה). הדוגמה הבאה ממחישה כיצד יש לקרוא את הערכים שבמאפיינים **X** ו-**Y** שבמבנה **ScreenPosition**:

```
ScreenPosition origin = new ScreenPosition(0, 0);
int xpos = origin.X;
int ypos = origin.Y;
```

שים לב שהגישה למאפיינים ושדות מבוצעת באמצעות תחביר זהה. כאשר המאפיין מופיע בהקשר של קריאה, המהדר לוקח את הקוד ומתרגם אותו באופן אוטומטי לאחראי הגישה **get** של המאפיין. כאשר המאפיין נמצא בהקשר של כתיבה, המהדר לוקח את הקוד ומתרגם אותו באופן אוטומטי לאחראי הגישה **set** של המאפיין:

```
origin.X = 40;
origin.Y = 100;
```

הערכים המוצבים מועברים לאחראי הגישה **set** באמצעות המשתנה **value**, כפי שצוין קודם. סביבת ההרצה מבצעת את ההעברה באופן אוטומטי.

ניתן גם להשתמש במאפיין בהקשר משולב של קריאה וכתיבה. במקרה זה, שני האחראים לגישה, **get** ו-**set**, יופעלו. לדוגמה, את המשפט הבא המהדר יתרגם אוטומטית לקריאות לאחראי הגישה **get** ו-**set**:

```
origin.X += 10;
```

טיפ

ניתן להכריז על מאפיינים **סטטיים**, באופן זהה להכרזה על שדות **סטטיים** ושיטות סטטיות. הגישה למאפיינים סטטיים מתבצעת באמצעות שם המחלקה או המבנה, ולא באמצעות שם המופע של המחלקה או המבנה.

מאפיינים לקריאה בלבד

ניתן להכריז על מאפיינים המכילים רק אחראי גישה **get**. במקרים אלה, ניתן להשתמש במאפיין רק בהקשר של קריאה. לדוגמה, להלן המאפיין **X** מהמבנה **ScreenPosition**, אשר מוצג כמאפיין קריאה בלבד:

```
struct ScreenPosition
{
   ...
   public int X
   {
      get { return this.x; }
   }
}
```

המאפיין **X** אינו כולל אחראי גישה **set**. על כן, כל ניסיון להשתמש בו בהקשר של כתיבה יגרום לשגיאה בשלב ההידור. לדוגמה:

```
origin.X = 140; // compile-time error
```

מאפיינים לכתיבה בלבד

ניתן גם להכריז על מאפיינים המכילים אחראי גישה **set** בלבד. במקרים אלה, ניתן להשתמש במאפיין רק בהקשר של כתיבה. לדוגמה, להלן המאפיין **X** מהמבנה **ScreenPosition**, אשר מוצג כמאפיין כתיבה בלבד:

```
struct ScreenPosition
{
   ...
   public int X
   {
      set { this.x = rangeCheckedX(value); }
   }
}
```

המאפיין **X** אינו כולל אחראי גישה **get**. לכן, כל ניסיון להשתמש בו בהקשר של קריאה יגרום לשגיאה בשלב ההידור. לדוגמה:

```
Console.WriteLine(origin.X);      // compile-time error
origin.X = 200;                   // compiles ok
origin.X += 10;                   // compile-time error
```

הערה

מאפיינים לכתיבה בלבד שימושים מאוד לאחסון של מידע מאובטח, כמו סיסמאות. יישום מאובטח אמור לאפשר לך לקבוע את הסיסמה, אך לעולם אסור לו לאפשר לך לקרוא אותה. שיטת התחברות (login) צריכה להשוות בין מחרוזת שהוקלדה על ידי המשתמש לסיסמה המאוחסנת, ולהחזיר איתות כלשהו אם השתיים שוות זו לזו.

נגישות של מאפיינים

הנגישות של המאפיין (ציבורי, פרטי או מוגן) נקבעת בעת ההכרזה על המאפיין. עם זאת, ניתן לקבוע רמות נגישות שונות עבור אחראי הגישה get ו-set. לדוגמה, בגרסה למבנה ScreenPosition המוצגת להלן, אחראי הגישה set של המאפיינים X ו-Y הוגדרו כפרטיים (אחראי הגישה get נשארים ציבוריים).

```
struct ScreenPosition
{
    ...
    public int X
    {
        get { return this.x; }
        private set { this.x = rangeCheckedX(value); }
    }
```

```
    public int Y
    {
        get { return this.y; }
        private set { this.y = rangeCheckedY(value); }
    }
    ...
    private int x, y;
}
```

בעת הגדרת נגישויות שונות מאלו של המאפיין עבור אחראי הגישה, עליך לקיים מספר כללים:

- ניתן לשנות בעת ההגדרה את הנגישות של אחד מאחראי הגישה בלבד. אין היגיון בהכרזה על מאפיין **כציבורי**, אם אתה מתכוון לשנות את הנגישות של שני אחראי הגישה שלו **לפרטית**.
- לא ניתן להגדיר נגישות מחמירה פחות מזו של המאפיין. לדוגמה, אם המאפיין מוגדר **כפרטי**, אינך יכול לשנות את הנגישות של אחראי הגישה **לציבורית**. אם בכל זאת ברצונך לעשות זאת, עליך להגדיר את המאפיין **כציבורי**, ולשנות את הנגישות של אחד מאחראי הגישה **לפרטית**.

הבנת מגבלות המאפיין

מאפיינים נראים ומתנהגים כמו שדות, אולם הם אינם שדות, וחלים עליהם מספר מגבלות :

- לא ניתן לאתחל מאפיין של מבנה או של מחלקה באמצעות אחראי גישה **set**. הקוד בדוגמה הבאה אינו חוקי, מכיוון שהמשתנה location לא אותחל (באמצעות **new**):

```
ScreenPosition location;
location.X = 40; // compile-time error, location not assigned
```

הערה

ייתכן שהדבר יישמע לך טריוויאלי, אבל אם X היה שדה ולא מאפיין, לא היה כל קושי עם הקוד שלעיל (זכור כי ScreenPosition הוא מבנה ולא מחלקה). מכאן משתמע שיש הבדלים בין שדות לבין מאפיינים. לכן, מלכתחילה רצוי להגדיר את המבנים והמחלקות באמצעות מאפיינים, ולא רצוי להשתמש בשדות אשר מאוחר יותר תצטרך להפוך למאפיינים. הקוד אשר משתמש במבנים ובמחלקות הללו עשוי שלא לפעול יותר כאשר תשנה את השדות למאפיינים.

- אינך יכול להשתמש במאפיין בתור ארגומנט **ref** או **out**, למרות ששדה נגיש לכתיבה יכול לשמש כארגומנט **ref** או **out**. לדוגמה:

```
MyMethod(ref location.X); // compile-time error
```

- מאפיין יכול להכיל לכל היותר אחראי גישה **get** אחד ואחראי גישה **set** אחד. מאפיין אינו יכול להכיל שיטות, שדות או מאפיינים נוספים.

- אחראי הגישה **get** ו-**set** אינם יכולים לקבל פרמטרים. הנתון המוצב מועבר לאחראי הגישה **set** באופן אוטומטי באמצעות משתנה **value**.
- לא ניתן להכריז על מאפייני **const** או **readonly**. לדוגמה:

```
const int X { get { ... } set { ... } } // compile-time error
```

הערה

כדי שמאפיין יהיה נגיש לקריאה בלבד, עליך רק להשמיט את אחראי הגישה **set**.

שימוש נכון במאפיינים

המאפיינים הם כלי תכנות בעל עוצמה, בעל תחביר נקי ומקביל לזה של השדות. ניצול נבון של המאפיינים תורם ליצירת קוד קל להבנה ולאחזקה. עם זאת, המאפיינים אינם מהווים תחליף לעיצוב מונחה-עצמים זהיר, אשר מתמקד בהתנהגות של האובייקטים ולא במאפייניהם. גישה לשדות פרטיים באמצעות שיטות רגילות או באמצעות מאפיינים אינה מספיקה ליצירת קוד מעוצב היטב. לדוגמה, בחשבון הבנק קיימת יתרה, ולכן אתה עשוי להתפתות ליצור מאפיין **Balance** (יתרה) במחלקה **BankAccount**, באופן הבא:

```
class BankAccount
{
    ...
    public money Balance
    {
        get { ... }
        set { ... }
    }
    private money balance;
}
```

זהו עיצוב שאינו נכון, מכיוון שאינו מייצג את הפעולות המתבקשות בעת הפקדה ומשיכה של כסף לחשבון, שהן פעולות רגילות בחשבון בנק. בעת תכנות, השתדל לא לאבד את המשמעות של הבעיה בתוך הסבך של תחביר הקוד. נסה להביע את הבעיה שעליך לפתור בצורה ברורה:

```
class BankAccount
{
   ...
   public money Balance { get { ... } }
   public void Deposit(money amount) { ... }
   public bool Withdraw(money amount) { ... }
   private money balance;
}
```

הכרזה על מאפיינים בממשק

על **ממשקים** (Interfaces) למדנו בפרק 12. ממשקים יכולים להכיל גם מאפיינים. כדי לעשות זאת, עליך להכריז על מילות המפתח **set** או **get**, או על שתיהן, אך להחליף את גוף אחראי הגישה (accessor) בנקודה-פסיק. לדוגמה:

```
interface IScreenPosition
{
   int X { get; set; }
   int Y { get; set; }
}
```

המבנה או המחלקה אשר מממשים את הממשק חייבים לממש גם את אחראי הגישה. לדוגמה:

```
struct ScreenPosition : IScreenPosition
{
   ...
   public int X
   {
      get { ... }
      set { ... }
   }
   public int Y
   {
      get { ... }
      set { ... }
   }
   ...
}
```

אם אתה מממש את מאפייני הממשק במחלקה, תוכל להגדיר את מימוש המאפיין **כווירטואלי**, וכך לאפשר למחלקות נגזרות עתידיות לעקוף את המימוש. לדוגמה:

```
class ScreenPosition : IScreenPosition
{
   ...
   public virtual int X
   {
      get { ... }
      set { ... }
   }
   public virtual int Y
   {
      get { ... }
      set { ... }
   }
   ...
}
```

הערה

דוגמה זו מציגה מחלקה. זכור שמילת המפתח **virtual** אינה תקפה במבנים (sturcts) מכיוון שלא ניתן לגזור ממבנים. מבנה לעולם יהיה **חתום** (sealed).

באפשרותך לממש מאפיין גם באמצעות התחביר של **מימוש ממשק מפורש** (explicit interface implementation, ראה פרק 12). מימוש מפורש של מאפיין אינו ציבורי ואינו וירטואלי, ולכן לא ניתן לעקוף אותו. לדוגמה:

```
struct ScreenPosition : IScreenPosition
{
   ...
   int IScreenPosition.X
   {
      get { ... }
      set { ... }
   }
   int IScreenPosition.Y
   {
      get { ... }
      set { ... }
   }
   ...
   private int x, y;
}
```

שימוש במאפיינים ביישום Windows

כאשר אתה משתמש בחלון Properties של Microsoft Visual Studio 2005, אתה יוצר למעשה קוד אשר מציב ומושך את ערכי המאפיינים של רכיבי יישומים שונים, כגון פקדי **TextBox**, **טפסים** (**Forms**) **ולחצנים** (**Buttons**). לחלק מהמרכיבים הללו יש מספר גדול של מאפיינים, וחלקם שימושיים יותר מאחרים. תוכל לשנות חלק גדול מהמאפיינים הללו בסביבת ההרצה באמצעות תחביר זהה לתחביר שלמדת בפרק זה.

בתרגיל הבא עליך להשתמש במספר מאפיינים מוגדרים מראש של פקדי **תיבת הטקסט** ושל המחלקה **Form**. כך ניתן ליצור יישום פשוט אשר מציג תמיד את גודל החלון שלו, גם לאחר שינוי בגודל. כמו כן, תצטרך להציג את גודלו בכותרת שלו.

השתמש במאפיינים

1. הפעל את סביבת הפיתוח Visual Studio 2005.
2. פתח את הפרויקט **Properties** שבתיקייה My Documents\Microsoft Press\Visual CSharp Step by Step\Chapter 14\Properties.

3. בתפריט Debug בחר Start Without Debugging.

הפרויקט ייבנה ויופעל. טופס Windows יציג שני תיבות טקסט ריקות בשם Width ו-Height (רוחב וגובה).

בתוכנית עצמה, תיבות הטקסט נקראות גם הן **width** ו-**height**. לתיבות הטקסט יש מאפיינים רבים, אך בתרגיל זה השתמש במאפיין **Text**, שנמצא בו הכיתוב המוצג בתיבת הטקסט.

4. סגור את הטופס וחזור לסביבת הפיתוח.

5. הצג את קובץ המקור Form1.cs בחלון Code and Text Editor. אתר את השיטה **resize** הראשונה המכילה את הערה to do //.

שיטה זו מופעלת על ידי הבנאי Form1. עליך להשתמש בה כדי להציג את הגודל הנוכחי של הטופס בתיבות הטקסט **width** ו-**height**. הרוחב ההתחלתי של הטופס הוא 232 פיקסלים, והגובה שלו הוא 96 פיקסלים.

6. הוסף שני משפטים לשיטה **resize** (שינוי גודל) שתפקידם להציג את גודל הטופס. המשפט הראשון מציב את המחרוזת "232" במאפיין **Text** של תיבת הטקסט width. המשפט השני מציב את המחרוזת "96" במאפיין **Text** של תיבת הטקסט height.

השיטה **resize** נראית כעת כך:

```
private void resize()
{
   width.Text = “232”;
   height.Text = “96”;
}
```

7. בתפריט Debug (ניפוי) בחר Start Without Debugging, כדי לבנות ולהפעיל את התוכנה.

שתי תיבות הטקסט שבטופס Windows יציגו את הערכים 232 ו-96. אחראי הגישה **set** של המאפיין **TextBox.Text** אחראי להצגת המחרוזות האמורות.

8. שנה את גודל הטופס. שים לב שהטקסט בתיבות הטקסט אינו משתנה.

עכשיו עליך להבטיח שהטקסט במאפיין **TextBox.Text** יציג תמיד את הגודל הנוכחי של הטופס.

9. סגור את הטופס.

10. הצג את Form1 בחלון Designer View.

11. לחץ Properties כדי להציג את מאפייני הטופס, ומצא את המאפיין Size.

שים לב שהמאפיין **Size** מכיל למעשה שני ערכים: הרוחב והגובה של הטופס. בסביבת ההרצה תוכל לקרוא ולכתוב את המאפיין **Size** של הטופס. הסוג של המאפיין **Size** הוא **struct** המכיל שני ערכים. אם תבדוק את המאפיין **Size** שבמחלקה **Form** שנמצאת ב-MSDN Library for Visual Studio 2005, תראה שהסוג של המאפיין נקרא גם כן Size, למרות הבלבול שעלול להיווצר מהכפילות:

```
struct Size
   {
   public int Height
   {
      get { ... }
      set { ... }
   }
   public int Width
   {
      get { ... }
      set { ... }
   }
}
```

ניתן לראות שהמבנה **Size** עצמו חושף את **width** ו-**height** כמאפיינים בפני עצמם. עליך לקרוא את המאפיין **Size.Width**, כדי לקבל את הרוחב הנוכחי של הטופס, ואת המאפיין **Size.Height** - כדי לקבל את הגובה הנוכחי שלו.

12. הצג את קובץ המקור Form1.cs בחלון Code and Text Editor וחזור לשיטה **resize** אשר ערכת קודם.

13. ערוך את שני המשפטים באופן שהערך **Size.Width** של הטופס יוצב במאפיין **Width.Text** ושהערך **Size.Height** של הטופס יוצב במאפיין **Height.Text**. עליך להמיר את הערך מסוג **int** לסוג **string**. הדרך הפשוטה ביותר לעשות זאת, הינה על ידי השיטה **ToString**.

השיטה **resize** נראית כעת כך:

```
private void resize()
{
   int w = this.Size.Width;
   width.Text = w.ToString();
   int h = this.Size.Height;
   height.Text = h.ToString();
}
```

14. בתפריט Debug בחר Start Without Debugging כדי לבנות ולהפעיל את התוכנית.

15. שנה את גודל הטופס Windows.

בזמן שאתה משנה את הגודל, תיבות הטקסט מתעדכנות כדי להציג את הגודל המשתנה. הקריאה לשיטה **resize** מתבצעת בכל פעם שגודל הטופס משתנה באמצעות האירוע Resize של הטופס. על **אירועים** (events) נלמד בפרק 16.

16. סגור את הטופס.

משימתך האחרונה היא להציג את גודל הטופס בכותרת שלו.

17. חזור לשיטה resize אשר ערכת קודם בטופס Form1.cs.

18. הוסף שני משפטים לשיטה **resize**. המשפט הראשון מפעיל את השיטה **string.Format** כדי ליצור מחרוזת בודדת המכילה את הרוחב והגובה של הטופס. המשפט השני מציב מחרוזת זו במאפיין **Text** הציבורי של הטופס.

הערה

השיטה הסטטית **string.Format** שימושית מאוד לבנייה ולסידור של מחרוזות מסוגי נתונים אחרים, כגון מספרים. היא פועלת בצורה דומה לשיטה **Consol.WriteLine**, אשר מסדרת מחרוזות כדי שיהיה ניתן להציגן על המסך.

השיטה resize נראית כעת כך:

```
private void resize()
{
   int w = this.Size.Width;
   width.Text = w.ToString();
   int h = this.Size.Height;
   height.Text = h.ToString();
   string s = string.Format("({0}, {1})", w, h);
   this.Text = s;
}
```

19. בתפריט Debug בחר Start Without Debugging.

הפרויקט ייבנה ויופעל. הכותרת תציג כעת את גודל הטופס, ותשתנה בכל פעם שתשנה את הגודל שלו.

20. סגור את הטופס וחזור לסביבת הפיתוח Visual Studio 2005.

☯ **אם ברצונך להמשיך לפרק הבא**

השאר את Visual Studio 2005 פעילה ועבור לפרק 15.

☯ **אם ברצונך לסגור כעת את Visual Studio 2005**

פתח את תפריט Menu ולחץ Exit. אם מופיעה תיבת דו-שיח Save, לחץ Yes.

פרק 14 – טבלה מסכמת

המשימה	צריך

להכריז על מאפיין קריאה או כתיבה עבור מבנה או מחלקה.	להכריז על סוג המאפיין, שמו, אחראי הגישה **get** ואחראי הגישה **set**. לדוגמה:
	`struct ScreenPosition`
	`{`
	`  ...`
	`  public int X`
	`  {`
	`  get { ... }`
	`  set { ... }`
	`  }`
	`  ...`
	`}`

להכריז על מאפיין לקריאה בלבד עבור מבנה או מחלקה.	להכריז על מאפיין המכיל אחראי גישה **get** בלבד. לדוגמה:
	`struct ScreenPosition`
	`{`
	`  ...`
	`  public int X`
	`  {`
	`  get { ... }`
	`  }`
	`  ...`
	`}`

להכריז על מאפיין לכתיבה בלבד עבור מבנה או מחלקה.	להכריז על מאפיין המכיל אחראי גישה **set** בלבד. לדוגמה:
	`struct ScreenPosition`
	`{`
	`  ...`
	`  public int X`
	`  {`
	`  set { ... }`
	`  }`
	`  ...`
	`}`

להכריז על מאפיין בממשק.	להכריז על מאפיין המכיל רק את מילות המפתח **set** או **get**, או את שתיהן. לדוגמה:
	`interface IScreenPosition`
	`{`
	`  int X { get; set; } // no body`
	`  int Y { get; set; } // no body`
	`}`

המשימה	צריך
לשלב מאפיין של ממשק במבנה או מחלקה.	במחלקה או במבנה בהם משלבים את הממשק, יש להכריז על המאפיין ולהוסיף את אחראי הגישה. לדוגמה:

```
struct ScreenPosition : IScreenPosition
{
    public int X
    {
        get { ... }
        set { ... }
        }
    public int Y
    {
        get { ... }
        set { ... }
    }
}
```

פרק 15

סדרנים

בסיום פרק זה, תוכל:

- **לכמס** (encapsulate) הכרזות לוגיות דמויות מערך באמצעות **סדרנים** (indexers).
- לשלוט בגישת הקריאה לסדרנים על ידי הכרזה על אחראי גישה **get**.
- לשלוט בגישת הכתיבה בסדרנים באמצעות הכרזה על אחראי גישה **set**.
- ליצור ממשק המכריז על סדרנים.
- לממש סדרנים במבנים ובמחלקות אשר יורשים ממשקים.

בפרק הקודם למדת להשתמש במאפיין ולממש מאפיינים כאמצעי לשליטה בגישה לשדות במחלקה. המאפיינים הינם כלי שימושי להצגת תמונת מראה (mirroring), או העתק מדויק, של שדות המכילים ערך בודד. כדי לגשת לפריטים המכילים מספר עצמים באמצעות תחביר מוכר ופשוט, אתה חייב להשתמש **בסדרנים** (indexers).

מהו סדרן?

סדרן (**indexer**) הוא מערך 'חכם' או מתוחכם, כשם שמאפיין הוא שדה 'חכם'. התחביר המשמש לעבודה עם סדרנים זהה לתחביר המשמש לעבודה עם מערכים. כדי להמחיש זאת, נשתמש בדוגמה. נבחן תחילה את הבעיה ונציג פתרון פשוט שאין בו שימוש בסדרנים. לאחר מכן ננסה להיעזר בסדרנים כדי למצוא פתרונות טובים יותר עבור אותה הבעיה. הבעיה עוסקת במספרים שלמים, או לשם דיוק – בסוג **int**.

פתרון ללא שימוש בסדרנים

סוג השלם **int** משמש בדרך כלל לאחסון ערך שלם (integer). למעשה, int מאחסן את הנתון כרצף של 32 סיביות שערך כל אחת מהן הוא 0 או 1. לרוב אין כל עניין בייצוג הבינארי הפנימי של המספר השלם, אלא רק משתמשים בסוג **int** לאחסון מספר שלם. לעיתים המתכנתים משתמשים בסוג **int** למטרה אחרת, כמו למשל שינוי סיביות מסוימות בתוך הערך **int** כדי לייצג עובדות או תנאים כלשהם (מתכנתים מרקע של C יודעים במה מדובר, כי הם מרבים להשתמש בדגלוני סימון). במילים אחרות, ייתכנו מקרים שבהם מתכנת ירצה להשתמש בסוג **int** מכיוון שהוא מכיל 32 סיביות, ולא בגלל היכולת לייצג מספר שלם.

הערה

יש מתכנתים ותיקים אשר משתמשים לעיתים בסוג **int** כדי לחסוך בזיכרון. הערך int מכיל 32 סיביות, שכל אחת מהן יכולה לקבל את הערך 0 או 1. במקרים מסוימים, מתכנתים מציבים 1 כדי לייצג ערך אמת (true), ומציבים 0 כדי לייצג ערך **שקר** (**false**), ולאחר מכן משתמשים ב-**int** כמערך של משתנים בוליאנים.

לדוגמה, הביטוי הבא משתמש באופרטורים >> ו-& (המשמשים לעריכת סיביות) כדי לדעת אם ערך הסיבית באינדקס 6 של הסוג **int** המכונה **bits**, הוא 0 או 1.

```
(bits & (1 << 6)) != 0
```

אם הסיבית שבאינדקס 6 (כלומר, מקום 6) היא 0, אז הערך של הביטוי כולו יהיה **false**. אם הסיבית שבאינדקס 6 היא 1, הערך של הביטוי כולו יהיה **true**. זהו ביטוי מורכב למדי, אולם הוא עדיין פשוט בהשוואה לביטוי הבא, אשר נועד להציב 0 בסיבית שבאינדקס 6:

```
bits &= ~(1 << 6)
```

או בהשוואה לביטוי הבא, אשר נועד להציב 1 בסיבית שבאינדקס 6:

```
bits |= (1 << 6)
```

למרות שהביטויים מבצעים את המוטל עליהם, הם בעייתיים, כי קשה להבין כיצד הם עושים זאת. הם מסובכים להפליא, והפתרון הוא ברמה נמוכה ביותר ואינו עוזר בהפשטה של הבעיה.

האופרטורים Bitwise ו-Shift

ייתכן שכמה מהסימנים בביטויים שבדוגמה שהוצגה לעיל אינם מוכרים לך, ובמיוחד הסימנים ~, >>, | ו-&. אלה הם חלק מהאופרטורים מסוג Bitwise (של סיביות) ומסוג Shift (הזזת סיביות), שמטרתם לבצע שינויים בסיביות יחידות המרכיבות נתונים מסוג **int** ו-**long**.

האופרטור ~ הוא אופרטור אונרי (unary) המבצע היפוך ברמת הסיבית. לדוגמה, אם תיקח מספר בן 8 סיביות שערכו 11001100 (204 בדצימלי) ותפעיל עליו את האופרטור ~, תקבל את הערך 00110011 (51 בדצימלי). זהו למעשה היפוך סיביות.

האופרטור >> הוא אופרטור בינארי המבצע הזזה לשמאל. הביטוי 2 >> 204 יחזיר את הערך 48 (בבינארי, הערך הדצימלי 204 שווה ל-11001100, והזזתו לשמאל בשני מקומות תיצור 00110000, או 48 בדצימלי). הסיביות בצד שמאל אובדות, ומהצד הימני נכנסים אפסים. ניתן גם להזיז לימין באותו אופן, באמצעות האופרטור <<. תוכל לנסות זאת ולראות שהספרות מימין אובדות ומשמאל נכנסים אפסים.

האופרטור | הוא אופרטור בינארי אשר מבצע פעולת OR ברמת הסיבית, ומחזיר 1 בכל מקום שבו לפחות אחד מהאופרנדים מכיל 1. לדוגמה, הביטוי 24 | 204 יחזיר את הערך 220 (204 הוא 11001100, 24 הוא 00011000 ו-220 הוא 11001110). נסה זאת.

האופרטור & מבצע פעולת AND ברמת הסיבית. אופרטור זה דומה לאופרטור | אולם הוא מחזיר 1 בכל מקום שבו שני האופרנדים מכילים 1. לדוגמה, הביטוי 24 & 204 יחזיר את הערך 8 (204 הוא 11001100, 24 הוא 00011000 ו-8 הוא 00001000). נסה זאת.

האופרטור ^ מבצע פעולת XOR ברמת הסיבית, ומחזיר 1 כאשר רק אחת הסיביות מכילה 1, אבל לא שתיהן (כאשר שתי סיביות שתואמות במקום בשני האופרנדים מכילות 1, הערך המוחזר יהיה 0). לכן 24 ^ 204 שווה ל-212 (00011000 ^ 11001100 שווה ל-11010100). נסה זאת.

פתרון המשתמש בסדרנים

הבה נשכח לרגע את הפתרון הקודם כדי להיזכר שוב מהי בדיוק הבעיה הניצבת בפנינו. אנחנו רוצים להשתמש ב-**int** לא בתור מספר שלם, אלא בתור מערך של 32 סיביות. לפיכך, הדרך הטובה ביותר לפתור את הבעיה הזו היא להשתמש ב-**int** בדרך שבה היינו משתמשים במערך רגיל של 32 משתנים בוליאנים. או במילים אחרות, אם המשתנה **bits** הוא מסוג **int**, אנו רוצים את האפשרות לגשת לסיבית שבאינדקס 6 באופן הבא:

```
bits[6]
```

או, לדוגמה, להציב בסיבית שבאינדקס 6 את הערך **אמת** (**true**) באופן הבא:

```
bits[6] = true
```

למרבה הצער, לא ניתן להשתמש בסימון סוגריים מרובעים בסוג **int**. סימון זה פועל רק על מערכים או סוגים המתנהגים כמו מערכים. כלומר, על סוגים המכריזים על סדרן (indexer). לכן, הפתרון לבעיה שלנו יהיה ליצור סוג משתנה חדש אשר מתנהג, נראה ומופעל בדיוק כמו מערך של משתנים מסוג **bool**, אולם ממומש באמצעות הסוג **int**. לסוג החדש נקרא **IntBits**. לפיכך **IntBits** יכיל ערך **int** (אשר יאותחל במסגרת הבנאי שלו), אולם הרעיון הוא להשתמש בסוג **IntBits** כמערך של משתנים מסוג **bool**.

טיפ

מכיוון שהסוג **IntBit** קטן ביותר, עדיף להגדירו כמבנה (struct) ולא כמחלקה.

```
struct IntBits
{
   public IntBits(int initialBitValue)
   {
      bits = initialBitValue;
   }
   // indexer to be written here
   private int bits;
}
```

כדי להגדיר את הסדרן, עליך להשתמש בסימון שהוא הכלאה בין מאפיין למערך. הסדרן של המבנה IntBits נראה כך:

```
struct IntBits
{
   ...
```

```
    public bool this [ int index ]
    {
        get
        {
            return (bits & (1 << index)) != 0;
        }
        set
        {
            if (value) // Turn the bit on if value is true, otherwise
            turn it off
                bits |= (1 << index);
            else
                bits &= ~(1 << index);
        }
    }
    ...
}
```

חשוב לשים לב לנקודות הבאות:

- סדרן אינה שיטה (method). אין לו סוגריים רגילים, אבל יש לו סוגריים מרובעים.
- סדרן מקבל תמיד ארגומנט יחיד שמוצג בין סוגריים מרובעים. מטרת הארגומנט היא לציין את מספר המרכיב שאליו מתבצעת הגישה.
- כל הסדרנים משתמשים במילת המפתח **this** במקום שם השיטה. למחלקה או למבנה מותר להגדיר סדרן אחד בלבד, ושמו חייב להיות **this**.
- סדרנים, בדומה למאפיינים, כוללים אחראי גישה **set** ו-**get** (accessors). אלה יכילו את הביטויים המסובכים ברמת הסיביות אשר הצגנו קודם לכן.
- הארגומנט שמופיע בהכרזה על הסדרן מכיל את ערך האינדקס המופיע בעת הקריאה לסדרן. אחראי הגישה **set** ו-**get** יכולים לקרוא את הארגומנט הזה כדי לדעת לאיזה מרכיב עליהם לגשת.

הערה

עליך לבצע בדיקת טווח עבור ערך האינדקס שמקבל הסדרן כדי למנוע הופעת חריגים בלתי צפויים בקוד של הסדרן.

בסיום ההכרזה על הסדרן ניתן להשתמש במשתנים מסוג **IntBits** במקום **int**, וליישם את סימון הסוגריים המרובעים על פי הצורך.

```
int adapted = 63;
IntBits bits = new IntBits(adapted);
bool peek = bits[6];       // retrieve bool at index 6
bits[0] = true;            // set the bit at index 0 to true
bits[31] = false;          // set the bit at index 31 to false
```

קל להיווכח שהתחביר החדש קל הרבה יותר להבנה. תחביר זה מבטא באופן ישיר וקולע אל תמצית הבעיה.

הערה

סדרנים ומאפיינים דומים זה לזה מהבחינה ששניהם עושים שימוש באחראי הגישה **get** ו-**set**. סדרן הוא למעשה מאפיין בעל מספר רב של ערכים. עם זאת, למרות שמותר להכריז על מאפיינים **סטטיים** (**static**), סדרנים סטטיים אינם חוקיים.

אחראי הגישה של הסדרנים

כאשר אתה קורא מהסדרן (indexer), המהדר מתרגם באופן אוטומטי את הקוד דמוי-המערך לקריאה לאחראי הגישה **get** (get accessor) של הסדרן. ראה לדוגמה את הקוד הבא:

```
bool peek = bits[6];
```

משפט זה יומר לקריאה לאחראי הגישה **get** של **bits**, והערך של הארגומנט **index** יהיה 6.

בנוסף, בעת כתיבה לסדרן, המהדר מתרגם באופן אוטומטי את הקוד דמוי-המערך לקריאה לאחראי הגישה **set** של הסדרן, ומציב את הערך הדרוש בארגומנט **index**. ראה לדוגמה את המשפט הבא:

```
bits[6] = true;
```

משפט זה יוצר קריאה לאחראי הגישה **set** של **bits** עם ערך **index** של 6. בדומה למאפיינים רגילים, הערך שתזין לסדרן (במקרה זה **true**) עובר לאחראי הגישה באמצעות מילת המפתח **value**. סוג הערך זהה לסוג הסדרן עצמו (במקרה זה **bool**).

ניתן להשתמש בסדרן בהקשר קריאה/כתיבה משולב. במקרה כזה מופעלים שני אחראי הגישה (**get** ו-**set**). קח לדוגמה את המשפט הבא:

```
bits[6] ^= true;
```

משפט זה יתורגם באופן אוטומטי לקוד הבא:

```
bits[6] = bits[6] ^ true;
```

קוד זה פועל מכיוון שהסדרן מכיל גם אחראי גישה **set** וגם אחראי גישה **get**.

הערה

ניתן להכריז על סדרן המכיל אחראי גישה **get** בלבד (סדרן לקריאה-בלבד), או אחראי גישה **set** בלבד (סדרן לכתיבה-בלבד).

השוואה בין סדרנים למערכים

אין זה מקרי שהתחביר המשמש לשליטה בסדרן (indexer) דומה מאוד לתחביר המשמש לשליטה במערך (Array). עם זאת, יש גם מספר הבדלים חשובים ביניהם:

- סדרנים יכולים להשתמש בסימון תחתי (subscript) שאינו נומרי, בעוד שמערכים יכולים להשתמש בסימון תחתי שהינו מספר שלם (integer) בלבד:

```
public int this [ string name ] { ... } // okay
```

טיפ

מחלקות אוסף (collection classes) רבות, כגון **Hashtable**, אשר ממשות חיפוש אסוציאטיבי המתבסס על צמדי מפתח-ערך, מפעילים סדרנים כחלופה נוחה לשיטה **Add** להוספת ערכים חדשים, ולדפדוף במאפיין **Values** כדי למצוא ערכים בקוד. לדוגמה, במקום הקוד הזה:

```
Hashtable ages = new Hashtable();
ages.Add("John", 41);
```

ניתן להשתמש בקוד הבא:

```
Hashtable ages = new Hashtable();
ages["John"] = 41;
```

- ניתן להעמיס (overload) סדרנים בדומה להעמסת שיטות, אבל לא ניתן להעמיס מערכים:

```
public Name this [ PhoneNumber number ] { ... }
public PhoneNumber this [ Name name ] { ... }
```

- לא ניתן להשתמש בסדרנים כפרמטרים מסוג ref או out, אבל ניתן לעשות זאת עם מרכיבי המערכים:

```
IntBits bits;              // bits contains an indexer
Method(ref bits[1]);       // compile-time error
```

מאפיינים, מערכים וסדרנים

מאפיין יכול להחזיר מערך, אולם עליך לזכור שמערכים הם מסוג הפניה, ולכן חשיפה של מערך בתור מאפיין עשויה לגרום לשכתוב לא מכוון של נתונים. המבנה (struct) הבא גורם לחשיפת מאפיין מערך בשם **Data**:

```
struct Wrapper
{
   int[] data;
   ...
   public int[] Data
   {
      get { return this.data; }
      set { this.data = value; }
   }
}
```

כעת עיין בקוד הבא אשר מפעיל את המאפיין Data:

```
Wrapper wrap = new Wrapper();
...
int[] myData = wrap.Data;
myData[0]++;
myData[1]++;
```

הקוד נראה לכאורה בלתי מזיק. אולם, מכיוון שמערכים הם מסוג הפניה, המשתנה **myData** מתייחס לאותו אובייקט שאליו מפנה המשתנה הפרימיטיבי **data** שבמבנה **Wrapper**. כל שינוי במרכיבים של **myData** יוחל גם על המרכיבים של המערך **data**. למשפט ++myData[0] תהיה תוצאה זהה לחלוטין לזו של המשפט ++data[0]. אם אין זו הכוונה, ניתן גם להשתמש בשיטה **Clone** שבתחום אחראי הגישה **get** של המאפיין **Data**, כדי להחזיר עותק של המערך data, אך זה עשוי להיות מורכב ויקר מבחינת שימוש בזיכרון. על ידי סדרנים ניתן לכתוב פתרון טבעי לבעיה זו - במקום לחשוף את המערך כולו כמאפיין, עליך רק לגרום לכל אחד ממרכיביו להיות זמין באמצעות הסדרן:

```
struct Wrapper
{
   int[] data;
   ...
   public int this [int i]
   {
      get { return this.data[i]; }
      set { this.data[i] = value; }
   }
}
```

הקוד שלהלן מפעיל סדרן בדרך דומה להפעלת המאפיין שראינו בדוגמה הקודמת:

```
Wrapper wrap = new Wrapper();
...
int[] myData = new int[2];
myData[0] = wrap[0];
myData[1] = wrap[1];
myData[0]++;
myData[1]++;
```

הפעם, הגדלת הערכים במערך **MyData** אינה משפיעה כלל על המערך המקורי שבאובייקט **Wrapper**. אם תרצה לשנות את המידע שבאובייקט **Wrapped**, תצטרך לכתוב משפט כזה:

```
wrap[0]++;
```

פתרון זה "נקי", פשוט ובטוח הרבה יותר!

סדרנים בממשקים

ניתן להכריז על סדרן במסגרת ממשק. כדי לעשות זאת, כתוב את מילות המפתח **get** ו/או **set**, אך החלף את הגוף של מילות המפתח בנקודה-פסיק. המחלקה או המבנה אשר מממשים את הממשק צריכים לממש גם את אחראי הגישה של הסדרן שהוכרז בממשק. לדוגמה:

```
interface IRawInt
{
   bool this [ int index ] { get; set; }
}
struct RawInt : IRawInt
{
   ...
   public bool this [ int index ]
   {
      get { ... }
      set { ... }
   }
   ...
}
```

אם אתה מממש את הממשק במחלקה, תוכל להכריז על מימושי הסדרן כווירטואליים (virtual). הדבר יאפשר למחלקות נגזרות עתידיות לעקוף את אחראי הגישה **get** ו-**set**. לדוגמה:

```
class RawInt : IRawInt
{
   ...
   public virtual bool this [ int index ]
   {
      get { ... }
      set { ... }
   }
   ...
}
```

באפשרותך לממש סדרנים גם באמצעות התחביר של מימוש ממשק מפורש (explicit interface implementation. ראה פרק 12). מימוש מפורש של סדרן אינו ציבורי ואינו וירטואלי, ולכן לא ניתן לעקוף אותו. לדוגמה:

```
struct RawInt : IRawInt
{
   ...
   bool IRawInt.this [ int index ]
   {
      get { ... }
      set { ... }
   }
   ...
}
```

שימוש בסדרנים ביישומי Windows

בתרגיל הבא עליך ללמוד יישום פשוט של ספר טלפונים ולהשלים את המימושים החסרים בו. משימתך היא לכתוב שני סדרנים במחלקה **PhoneBook**: אחד מקבל פרמטר מסוג **Name** ומחזיר סוג **PhoneNumber**, ואחד מקבל פרמטר מסוג **PhoneNumber** ומחזיר סוג **Name**. כדי להקל עליך, המבנים **Name** ו-**PhoneNumber** כבר נכתבו עבורך. בנוסף, עליך לקרוא לסדרן מהמקומות הנכונים בתוכנית.

הכרת היישום

1. הפעל את Microsoft Visual Studio 2005.
2. פתח את הפרויקט **Properties** שבתיקייה My Documents\Microsoft Press\Visual CSharp Step by Step\Chapter 15\Indexers.

   זהו יישום Microsoft Windows Forms.
3. בתפריט Debug בחר Start Without Debugging.

   הפרויקט ייבנה ויופעל. הטופס שיוצג מכיל שתי תיבות טקסט ריקות המסומנות ב-Name ו- Phone Number. בטופס יש גם שלושה לחצנים - אחד משמש להוספת הצמד שם/מספר טלפון לרשימת השמות ומספרי הטלפון הכלולה ביישום; השני משמש למציאת מספר טלפון לפי שם; והשלישי משמש למציאת שם באמצעות מספר טלפון. נכון לעכשיו, לחצנים אלה אינם מבצעים אף לא אחת מהפעולות. משימתך היא לגרום ללחצנים הללו לפעול כנדרש.
4. סגור את הטופס וחזור לסביבת הפיתוח.
5. הצג את קובץ המקור Name.cs בחלון Code and Text Editor. הבט במבנה Name, שמטרתו לאחסן שמות.

   השם מועבר לבנאי בתור מחרוזת. ניתן למשוך את השם באמצעות מאפיין לקריאה בלבד בשם **Text**. השיטות **Equals** ו-**GetHashCode** משמשות להשוואה בין שמות בעת ביצוע חיפוש במערך, המורכב על ידי ערכים מסוג **Name**. התעלם מהן בשלב זה.
6. הצג את קובץ המקור PhoneNumber.cs בחלון Code and Text Editor, ועיין במבנה **PhoneNumber**. הוא דומה מאוד למבנה **Name**.
7. הצג את קובץ המקור PhoneBook.cs בחלון Code and Text Editor, ובחן את השיטה **PhoneBook**.

   מחלקה זו כוללת שני מערכים פרטיים: מערך של ערכים מסוג **Name** בשם **names**, ומערך של ערכים מסוג **PhoneNumber** בשם **phoneNumbers**. המחלקה **PhoneBook** מכילה גם שיטה **Add** שתפקידה להוסיף מספר טלפון ושם לספר הטלפונים. שיטה זו מופעלת כאשר המשתמש לוחץ על הלחצן **Add** שעל הטופס. השיטה Add קוראת לשיטה **enlargeIfFull** כדי לבדוק אם המערכים מלאים כאשר המשתמש רוצה להוסיף רשומה.

אם המערכים מלאים, השיטה יוצרת שני מערכים חדשים גדולים יותר, מעתיקה אליהם את הערכים שבמערכים הקיימים, ומוחקת את המערכים הקודמים, הקטנים.

כתיבת סדרנים

1. בקובץ המקור PhoneBook.cs הוסף למחלקה PhoneBook סדרן **ציבורי** (**public**) לקריאה בלבד, אשר מקבל פרמטר **PhoneNumber** ומחזיר **Name**. את גוף אחראי הגישה **get** השאר ריק.

הסדרן צריך להיראות כך:

```
sealed class PhoneBook
{
   ...
   public Name this [PhoneNumber number]
   {
      get
      {

      }
   }
   ...
}
```

2. ממש את אחראי הגישה **get**.

מטרת אחראי הגישה היא למצוא את השם התואם למספר הטלפון שהועבר כפרמטר. כדי לעשות זאת, עליך לקרוא לשיטה הסטטית **IndexOf** מהמחלקה **Array**. השיטה **IndexOf** מבצעת חיפוש במערך, ומחזירה את האינדקס של הפריט הראשון במערך התואם לערך החיפוש. הארגומנט הראשון שיש להעביר ל-**IndexOf** הוא המערך שיש לסרוק (**phoneNumbers**). הארגומנט השני שיש להעביר ל-**IndexOf** הוא הפריט שברצונך למצוא. השיטה **IndexOf** מחזירה את מספר האינדקס השלם של המרכיב אם הוא נמצא, ומחזירה 1- במקרה שהוא לא נמצא. אם הסדרן מאתר את מספר הטלפון, הוא יחזיר אותו; אחרת, עליו להחזיר ערך **Name** ריק. שים לב שהסוג **Name** הוא מבנה, ולכן תמיד יהיה לו בנאי ברירת מחדל אשר יאתחל את השדה הפרטי שלו ל-**null**.

לאחר מימוש אחראי הגישה **get**, הסדרן צריך להיראות כך:

```
sealed class PhoneBook
{
   ...
   public Name this [PhoneNumber number]
   {
      get
      {
         int i = Array.IndexOf(this.phoneNumbers, number);
         if (i != -1)
            return this.names[i];
         else
            return new Name();
      }
   }
...
}
```

3. הוסף למחלקה **PhoneBook** עוד סדרן ציבורי לקריאה בלבד, אשר מקבל פרמטר **Name** ומחזיר **PhoneNumber**. ממש את הסדרן הזה באותה הדרך שמימשת את הסדרן הראשון, כי גם **PhoneNumber** הוא מבנה ולכן גם הוא יכלול תמיד בנאי ברירת מחדל.

הסדרן השני צריך להיראות כך:

```
sealed class PhoneBook
{
   ...
   public PhoneNumber this [Name name]
   {
      get
      {
         int i = Array.IndexOf(this.names, name);
         if (i != -1)
            return this.phoneNumbers[i];
         else
            return new PhoneNumber();
      }
   }
   ...
}
```

שים לב שהסדרנים המועמסים יכולים להיות זמינים זה לצד זה, מכיוון שהחתימות שלהם שונות. אם היינו משתמשים במחרוזות פשוטות ולא במבנים **Name** ו-**PhoneNumber**, הסדרנים המועמסים היו בעלי חתימה זהה ולא היה ניתן להדר את המחלקה.

4. בתפריט Build בחר Build Solution. תקן שגיאות תחביר וצור שוב את התוכנית אם דרוש.

קריאה לסדרנים

1. הצג את קובץ המקור Form1.cs בחלון Code and Text Editor ואתר את השיטה **findPhone_Click**.

הקריאה לשיטה זו מתבצעת בעת לחיצה על לחצן Search הראשון (הקריאה לשיטה זו מתבצעת באמצעות אירועים (events) ונציגים (delegates), שתלמד עליהם בפרק 16). נכון לעכשיו שיטה זו ריקה. היא צריכה לבצע את הפעולות הבאות:

א. לקרוא את המחרוזת **Text** מתיבת הטקסט Name.

ב. אם המחרוזת איננה ריקה, יש לחפש את מספר הטלפון השייך לשם **בספר הטלפונים** ולהיעזר בסדרן (שים לב ש-**Form1** מכיל שדה **PhoneBook** פרטי בשם **phonebook**). אחר כך צריך לבנות אובייקט **Name** מהמחרוזת, ולהעביר אותו כפרמטר לסדרן **PhoneBook**.

ג. לכתוב את מספר הטלפון שהוחזר על ידי הסדרן בתיבת הטקסט **phoneNumber**.

2. כעת ממש את השיטה **findPhone_Click**.

השיטה צריכה להיראות כך:

```
partial class Form1 : System.Windows.Forms.Form
{
   ...
   private void findPhone_Click(object sender, System.EventArgs e)
   {
      string text = name.Text;
      if (text != “”)
      {
         phoneNumber.Text = phoneBook[new Name(text)].Text;
      }
   }
   ...
   private PhoneBook phoneBook = new PhoneBook();
}
```

.3 אתר את השיטה **findName_Click** בקובץ המקור Form1.cs. היא נמצאת אחרי השיטה **findPhone_Click**.

הקריאה לשיטה **findName_Click** מתבצעת בעת לחיצה על הלחצן Search השני. שיטה זו דומה לשיטה **findPhone_Click** ומטרותיה הן:

א. לקרוא את המחרוזת **Text** מתיבת הטקסט Phone Number.

ב. אם המחרוזת איננה ריקה, לחפש באמצעות הסדרן את השם אשר שייך למספר הטלפון **בספר הטלפונים**.

ג. לכתוב את **השם** שהוחזר על ידי הסדרן בתיבת הטקסט Name.

.4 כעת ממש את השיטה. השיטה צריכה להיראות כך:

```
partial class Form1 : System.Windows.Forms.Form
{
   ...
   private void findName_Click(object sender, System.EventArgs e)
   {
      string text = phoneNumber.Text;
      if (text != “”)
      {
         name.Text = phoneBook[new PhoneNumber(text)].Text;
      }
   }
   ...
}
```

.5 בתפריט Build בחר Build Solution.

הפעלת היישום

1. בתפריט Debug בחר Start Without Debugging.
2. הקלד את שמך ואת מספר הטלפון שלך לתיבות הטקסט, ולחץ Add.
לאחר שתלחץ Add, השיטה **Add** תאחסן את הרשומות בספר הטלפונים ותרוקן את תיבות הטקסט כדי שניתן יהיה לבצע בעזרתן חיפוש.
3. חזור על שלב 2 מספר פעמים עם שמות ומספרי טלפון שונים, כדי למלא את ספר הטלפונים במספר רשומות.
4. הקלד את אחד השמות שהזנת בשלב 2 בתיבת הטקסט Name, ולחץ כעת Search –>.
מספר הטלפון שהזנת בשלב 2 יישלף מספר הטלפונים ויוצג בתיבת הטקסט Phone Number.
5. מחק את השם שבתיבת הטקסט Name ולחץ Search <-.
השם יישלף מספר הטלפונים ויוצג בתיבת הטקסט Name.
6. הקלד שם שלא הכנסת לספר הטלפונים בתיבת הטקסט Name ולחץ כעת Search ->.
הפעם תיבת הטקסט Phone Number תישאר ריקה. כלומר, השם שהקלדת לא נמצא בספר הטלפונים.
7. סגור את הטופס.

☯ **אם ברצונך להמשיך לפרק הבא**

השאר את Visual Studio 2005 פעילה ועבור לפרק 16.

☯ **אם ברצונך לכבות כעת את Visual Studio 2005**

פתח את תפריט Menu ולחץ Exit. אם מופיעה תיבת דו-שיח Save, לחץ Yes.

פרק 15 - טבלה מסכמת

המשימה	צריך

להכריז על סדרן עבור מבנה או מחלקה.	להכריז על סוג הסדרן, לאחריו מילת המפתח **this**, ולאחר מכן הארגומנטים של הסדרן בין סוגריים מרובעים. גוף הסדרן יכול לכלול אחראי גישה **set** ו/או **get**. לדוגמה:
	`struct RawInt`
	`{`
	`  ...`
	`  public bool this [ int index ]`
	`  {`
	`  get { ... }`
	`  set { ... }`
	`  }`
	`  ...`
	`}`

להכריז על סדרן בממשק.	להכריז על סדרן המכיל רק את מילות המפתח **set** ו/או **get**. לדוגמה:
	`interface IRawInt`
	`{`
	`  bool this [ int index ] { get; set; }`
	`}`

לממש סדרן של ממשק במבנה או במחלקה.	במחלקה או במבנה בהם מממשים את הממשק, יש להכריז על הסדרן ולממש את אחראי הגישה. לדוגמה:
	`struct RawInt : IRawInt`
	`{`
	`  ...`
	`  public bool this [ int index ]`
	`  {`
	`  get { ... }`
	`  set { ... }`
	`  }`
	`  ...`
	`}`

לממש סדרן של ממשק באמצעות מימוש ממשק מפורש במחלקה או במבנה.	במחלקה או במבנה בהם מממשים את הממשק, לכתוב באופן מפורש את שם הסדרן, אך לא לציין את רמת הנגישות אליו. לדוגמה:
	`struct RawInt : IRawInt`
	`{`
	`  ...`
	`  bool IRawInt.this [ int index ]`
	`  {`
	`  get { ... }`
	`  set { ... }`
	`  }`
	`  ...`
	`}`

פרק 16

נציגים ואירועים

כאשר תסיים פרק זה, תוכל:

- להכריז על סוג נציג (delegate) כדי ליצור הפשטה של חתימת השיטה.
- ליצור מופע של נציג כדי להתייחס לשיטות מוגדרות ולשיטות אנונימיות.
- לקרוא לשיטה בעזרת נציג.
- הגדרת שדה אירוע.
- לטפל באירוע באמצעות נציג.
- להציף אירוע.

חלק גדול מהקוד שכתבת בתרגילים השונים שבספר זה מסתמך על העובדה שהמשפטים בתוכנית מופעלים בזה אחר זה. למרות שפעמים רבות זהו אכן המצב, לעיתים קרובות תצטרך להתערב ברצף הטבעי של הפעלת המשפטים, כדי לבצע פעולה חשובה יותר. בסיום הפעולה יכולה התוכנית לחזור לאותו המקום. הדוגמה הקלאסית לסוג כזה של תוכנית היא Windows Form. הטופס כולל פקדים שונים, כגון לחצנים ותיבות טקסט. כאשר המשתמש לוחץ על לחצן או מקליד טקסט בתיבת הטקסט, הטופס צריך להגיב מייד. כלומר, היישום צריך לחדול באופן זמני מפעולתו הנוכחית ולעבור לטיפול בקלט שזה עתה התקבל. אופן פעולה שכזה מאפיין לא רק ממשקים גרפיים, אלא כל יישום אשר מחייב תגובה מיידית (דוגמה מתחום אחר יכולה להיות כיבוי מיידי של כור אטומי במקרה של התחממות יתר).

הכרזה על נציגים ושימוש בהם

נציג (delegate) נראה ומתנהג בצורה דומה מאוד לזו של שיטה. בעת קריאה לנציג, סביבת ההרצה (runtime) מפעילה למעשה את השיטה שהנציג מפנה אליה. ניתן לשנות באופן דינמי את השיטה שהנציג מפנה אליה, ולכן ייתכן שהקוד אשר קורא לנציג יקרא לשיטה שונה בכל פעם שהוא מופעל. הדרך הטובה ביותר להבין מהם נציגים היא לראות אותם בפעולה, ולכן נשתמש בדוגמה.

הערה

אם עסקת בעבר בתכנות בשפת ++C, תוכל להשוות את הנציגים למצביעי פונקציה (function pointer). אולם, בניגוד למצביעי פונקציה, נציגים יכולים להפנות רק לשיטות בעלות חתימה זהה לשלהם, ולא ניתן לקרוא לנציג אשר אינו מפנה לשיטה תקפה.

תרחיש המפעל האוטומטי

נניח שעליך לכתוב מערכת בקרה למפעל אוטומטי. במפעל מספר רב של מכונות שונות. כל אחת מהמכונות מבצעת פעולה שונה כחלק מתהליך הייצור במפעל העוסק ביציקת לוחות מתכת, חיתוך לוחות, צביעת לוחות וכדומה. כל מכונה נבנתה והותקנה על ידי יצרן מתמחה אחר, כולן נשלטות באמצעות מחשב, וכל יצרן מספק למפעל קבוצת ממשקים (APIs) שמאפשרים לשלוט במכונות. משימתך היא לשלב בין כל מערכות השליטה השונות וליצור תוכנית בקרה מרכזית. אחד ממאפייני התוכנית צריך להיות היכולת לכבות את כל המכונות בבת-אחת, ובמידת הצורך - לעשות זאת באופן מיידי.

הערה

המונח API הוא **ממשק לתכנות יישום** (Application Programming Interface). זו שיטה, או קבוצת שיטות, שהשליטה עליהן מתבצעת דרך חלק מיישום. ניתן לראות ב- NET Framework קבוצה של ממשקי API, מכיוון שהיא מכילה שיטות המאפשרות לשלוט במערכת ההפעלה של סביבת ההרצה (common language runtime) ושל Microsoft Windows.

לכל מכונה יש תהליך ייחודי וגם API, אשר נשלטים על ידי המחשב, ובאמצעותם ניתן לכבות את המכונה בצורה בטוחה. להלן תמצית התהליכים הללו:

```
StopFolding();       // Folding and shaping machine
FinishWelding();     // Welding machine
PaintOff();   // Painting machine
```

מימוש תוכנית המפעל ללא שימוש בנציגים

לפניך גישה פשטנית למימוש אלמנט כיבוי המכונות בתוכנית:

```
class Controller
{
   ...
   public void ShutDown()
   {
      folder.StopFolding();
      welder.FinishWelding();
      painter.PaintOff();
   }
   ...
   // Fields representing the different machines
```

```
   private FoldingMachine folder;
   private WeldingMachine welder;
   private PaintingMachine painter;
}
```

גישה זו אומנם תצליח לכיבוי המכונות, אולם היא איננה גמישה ונוחה לשינוי. אם המפעל רוכש מכונה חדשה, תצטרך לשנות את הקוד. חסר קשר הדוק בין המחלקה **Controller** לבין המכונות השונות.

מימוש תוכנית המפעל בעזרת נציגים

למרות ששמות השיטות שונים זה מזה, לכל השיטות יש מבנה דומה. הן אינן מקבלות פרמטרים ואינן מחזירות ערך (בהמשך נלמד כיצד להתמודד עם מצבים שונים). לפניך המבנה הכללי של כל השיטות:

```
void methodName();
```

במקרים כאלה, כדאי להשתמש בנציגים (delegates). ניתן להשתמש בנציג המותאם למבנה של השיטות כדי להפנות לכל אחת מהשיטות המשמשות לכיבוי המכונות השונות. להלן אופן ההכרזה על נציג מסוג זה:

```
delegate void stopMachineryDelegate();
```

יש לשים לב לנקודות הבאות:

- בעת הכרזה על נציג צריך להשתמש במילת המפתח delegate.
- הנציג מגדיר את מבנה השיטות שהוא יכול להפנות אליהן. לכן, עליך לציין את סוג הערך המוחזר (void), את שם הנציג (stopMachineryDelegate) ואת הפרמטרים, אם יש (במקרה שלנו אין פרמטרים).

לאחר הגדרת הנציג, תוכל ליצור מופעים אל השיטות הרלוונטיות באמצעות האופרטור =+. תוכל לעשות זאת במסגרת הבנאי של המחלקה Controller באופן הבא:

```
class Controller
{
   delegate void stopMachineryDelegate();
   ...
   public Controller()
   {
      this.stopMachinery += folder.StopFolding;
   }
   ...
   // Create an instance of the delegate
   private stopMachineryDelegate stopMachinery;
}
```

דרוש מעט זמן להתרגל לתחביר הזה. אתה מוסיף (=+) את השיטה לנציג, ובשלב זה עדיין אינך קורא לשיטה. האופרטור + מועמס (overloaded) כדי שיקבל משמעות שונה כאשר הוא מופעל בהקשר של נציגים (נדון בהרחבה בנושא העמסת נציגים בפרק 19). שים לב שעליך לכתוב את שם השיטה בלבד, מבלי להוסיף סוגריים או פרמטרים.

ניתן להשתמש באופרטור =+ עם נציג לא מאותחל, מכיוון שבמקרה זה הנציג יאותחל אוטומטית. כמו כן, ניתן להשתמש במילת המפתח new כדי לאתחל נציג באופן מפורש אל שיטה מסוימת. הנה כך:

```
this.stopMachinery = new stopMachineryDelegate(folder.stopFolding);
```

ניתן גם לקרוא לשיטה באמצעות הפעלת הנציג באופן הבא:

```
public void ShutDown()
{
   this.stopMachinery();
   ...
}
```

הפעלת נציג נעשית באמצעות תחביר זהה לזה המשמש בעת קריאה לשיטה. אם השיטה שהנציג מפנה אליה מקבלת פרמטרים, עליך לציין אותם בעת הפעלתו.

הערה

כל ניסיון להפעיל נציג שאינו מאותחל יגרום לזריקת חריג **NullReference-Exception**.

היתרון העיקרי של השימוש בנציגים הוא ביכולת להפנות אל יותר משיטה אחת. כל שעליך לעשות זה להשתמש באופרטור =+ כדי להוסיפם לנציג באופן הבא:

```
public Controller()
{
this.stopMachinery += folder.StopFolding;
this.stopMachinery += welder.FinishWelding;
this.stopMachinery += painter.PaintOff;
}
```

הפעלת הנציג ()this.stopMachinery בשיטה **Shutdown** של המחלקה **Controller** תביא לקריאה אוטומטית לכל השיטות בזו אחר זו. השיטה **Shutdown** אינה צריכה לדעת את מספר המכונות הקיימות, או את שמות השיטות המשמשות לכבותן. ניתן להסיר שיטה מהנציג באמצעות האופרטור =-:

```
this.stopMachinery -= folder.StopFolding;
```

המבנה הנוכחי מוסיף את השיטות של המכונות השונות לנציג שבבנאי **Controller**. כדי לאפשר יצירת מחלקה **Controller** נפרדת לחלוטין מהמכונות השונות, עליך ליצור אמצעי המאפשר למחלקות שמחוץ למחלקה **Controller** להוסיף שיטות לנציג. יש מספר דרכים לעשות זאת:

- להגדיר את המשתנה StopMachinery של הנציג כציבורי (public):

```
public stopMachineryDelegate stopMachinery;
```

- להשאיר את המשתנה stopMachinery פרטי (private), אך להוסיף מאפיין קריאה/כתיבה המאפשר גישה אליו. בנוסף, עליך להגדיר את הסוג stopMachineryDelegate כציבורי:

```
public delegate void stopMachineryDelegate();
...
public stopMachineryDelegate StopMachinery
{
   get
   {
      return this.stopMachinery;
   }
   set
   {
      this.stopMachinery = value;
   }
}
```

- ליצור כימוס (encapsulation) מוחלט באמצעות מימוש של שיטות Add ו-Remove נפרדות. השיטה Add מקבלת שיטה בתור פרמטר ומוסיפה אותה לנציג, בשעה שהשיטה Remove מסירה את השיטה האמורה מהנציג (שים לב, שעליך להגדיר את השיטה בתור פרמטר באמצעות הסוג של הנציג):

```
public void Add(stopMachineryDelegate stopMethod)
{
   this.stopMachinery += stopMethod;
}
public void Remove(stopMachineryDelegate stopMethod)
{
   this.stopMachinery -= stopMethod;
}
```

אם אתה שואף ליצור תוכנית מוכוונת-עצמים, קרוב לוודאי שתעדיף להשתמש בשיטות Add/Remove. אולם במקרים רבים הדרכים האחרות הן חלופות שימושיות, ולכן הצגנו אותן כאן.

תהיה הטכניקה שתבחר אשר תהיה, עליך למחוק מהבנאי Controller את הקוד המוסיף את השיטה של המכונה לנציג. כעת תוכל ליצור **Controller** ואובייקטים אשר יכולים לייצג כל אחת מהמכונות בצורה הבאה. שים לב שבדוגמה זו השתמשנו בגישה Add/Remove:

```
Controller control = new Controller();
FoldingMachine folder = new FoldingMachine();
WeldingMachine welder = new WeldingMachine();
PaintingMachine painter = new PaintingMachine();
...
control.Add(folder.StopFolding);
control.Add(welder.FinishWelding);
control.Add(painter.PaintOff);
...
control.ShutDown();
...
```

שימוש בנציגים

בתרגיל הבא עליך ליצור נציג לכימוס שיטה אשר מציגה את השעה בתיבת טקסט של Microsoft Windows. עליך לשייך את אובייקט הנציג למחלקה Ticker שתפקידה להפעיל אותו בכל שנייה. כך תיצור יישום Windows המהווה שעון דיגיטלי פשוט.

השלם את יישום השעון הדיגיטלי

1. הפעל את Microsoft Visual Studio 2005.
2. פתח את הפרויקט **Delegates** שבתיקייה My Documents\Microsoft Press\Visual CSharp Step by Step\Chapter 16\Delegates.
3. בתפריט Debug בחר Start Without Debugging.

   כעת נוצר הפרויקט ומופעל. שעון דיגיטלי יוצג בטופס Windows. השעון אינו מכוון.
4. לחץ Start ולאחר מכן לחץ Stop.

   הלחיצות אינן משפיעות על היישום כי השיטות **Start** ו-**Stop** טרם מומשו. משימתך היא לממש את השיטות הללו.
5. סגור את החלון וחזור לסביבת התכנות של Visual Studio 2005.
6. פתח את קובץ המקור Ticker.cs והצג אותו בחלון Code and Text Editor. קובץ זה מכיל את המחלקה **Ticker** שמכילה את מנגנון הפעולה של השעון. השיטה משתמשת באובייקט מסוג **System.Timers.Timer** בשם **ticking**, כדי לבצע פעימה (pulse) בכל שנייה. המחלקה תופסת את הפעימה באמצעות אירוע (event) (על אירועים נלמד בהמשך פרק זה), ואחר כך דואגת לעדכון התצוגה באמצעות הפעלת נציג.
7. בחלון Code and Text Editor, אתר את ההכרזה של הנציג **Tick**. ההכרזה נמצאת סמוך לתחילת הקובץ ונראית כך:

```
public delegate void Tick(int hh, int mm, int ss);
```

ניתן להשתמש בנציג **Tick** כדי להפנות אל שיטה המקבלת שלושה פרמטרים שלמים מבלי להחזיר ערך. בסוף המחלקה מופיע משתנה נציג **tickers**, המתבסס על סוג זה. השיטות **Add** ו-**Remove** שבמחלקה זו מאפשרות לשיטות מתאימות להוסיף ולהסיר את עצמן ממשתנה הנציג **ticker**.

```
class Ticker
{
   ...
   public void Add(Tick newMethod)
   {
      this.tickers += newMethod;
   }
   public void Remove(Tick oldMethod)
   {
```

```
        this.tickers -= oldMethod;
    }
    ...

    private Tick tickers;
}
```

8. פתח את קובץ המקור Clock.cs בחלון Code and Text Editor. המחלקה **Clock** אחראית לעיצוב תצוגת השעון. היא מכילה את השיטות **Start** ו–**Stop** אשר ישמשו להפעלה ולהפסקה של השעון (לאחר שתיצור אותן), ומחלקה בשם **RefreshTime** אשר יוצרת מחרוזת המייצגת את הזמן הנגזר משלושת הפרמטרים שהשיטה מקבלת (שעות, דקות ושניות) ומציגה אותה בשדה **display** מסוג **TextBox**. שדה זה מאותחל במסגרת בנאי. המחלקה כוללת גם שדה **Ticker** פרטי בשם **pulsed**, שתפקידו להודיע לשעון מתי להתעדכן:

```
class Clock
{
    ...
    public Clock(TextBox displayBox)
    {
        this.display = displayBox;
    }
    ...
    private void RefreshTime(int hh, int mm, int ss )
    {
        this.display.Text = string.Format("{0:D2}:{1:D2}:{2:D2}",
        hh, mm, ss);
    }

    private Ticker pulsed = new Ticker();
    private TextBox display;
}
```

9. הצג את הקוד של קובץ המקור Form1.cs בחלון Code and Text Editor. שים לב שהבנאי יוצר מופע חדש של המחלקה **Clock** ומעביר לה את **שדה תיבת הטקסט digital** בתור פרמטר.

```
public Form1()
{
...
clock = new Clock(digital);
}
```

השדה **digital** הוא הפקד **TextBox** אשר מופיע בטופס. בתיבת טקסט זו יוצג הפלט המתקבל מהשעון.

10. חזור לקובץ המקור Clock.cs. צור את השיטה **Clock.Start** כך שתוסיף את השיטה **Clock.RefreshTime** לנציג שבאובייקט pulsed באמצעות השיטה **Ticker.Add**.

השיטה **Start** נראית כעת כך:

```
public void Start()
{
   pulsed.Add(this.RefreshTime);
}
```

11. ממש את השיטה **Clock.Stop** כך שתסיר את השיטה **Clock.RefreshTime** מהנציג באובייקט pulsed באמצעות השיטה **Ticker.Remove**.

השיטה **Stop** נראית כעת כך:

```
public void Stop()
{
   pulsed.Remove(this.RefreshTime);
}
```

12. בתפריט Debug בחר Start Without Debugging. הפרויקט ייבנה ויופעל.

13. לחץ Start.

הטופס Windows מציג כעת את השעה הנכונה ומתעדכן בכל שנייה.

14. לחץ Stop.

התצוגה מפסיקה להגיב ו"קופאת". הסיבה לכך היא שהלחצן Stop קורא לשיטה **Clock.Stop**, אשר מסירה את השיטה **RefreshTime** מהנציג **Ticker**, ולכן לא מתבצעת יותר קריאה לשיטה בכל שנייה.

15. לחץ Start.

השעון ימשיך את פעולתו ויתעדכן בכל שנייה. הסיבה לכך היא שהלחצן Start קורא לשיטה **Clock.Start**, שמוסיפה שוב את השיטה **RefreshTime** לנציג **Ticker**.

16. סגור את הטופס.

שיטות אנונימיות ונציגים

כל דוגמאות הוספת שיטה לנציג אשר הצגנו עד כה משתמשות בשם השיטה. נשתמש כעת שוב בדוגמת המפעל האוטומטי שהוצגה קודם. כדי להוסיף את השיטה **StopFolding** של האובייקט folder לנציג **stopMachinery**, השתמשנו בקוד הבא:

```
this.stopMachinery += folder.StopFolding;
```

דרך זו שימושית ביותר אם חתימת השיטה זהה לחתימה של הנציג, אולם מה קורה אם שיטה זו אינה קיימת? נניח שזו הייתה החתימה של השיטה **StopFolding**:

```
// Shut down within the specified number of seconds
void StopFolding(int shutDownTime);
```

כעת חתימת השיטה שונה מהחתימות של השיטות **FinishWelding** ו-**PaintOff**, ולפיכך איננו יכולים להשתמש בנציג אחד עבור שלושתן.

יצירת מתאם שיטות

כדי לפתור את הבעיה וליצור מתאם אחד עבור כמה שיטות, עלינו ליצור שיטה נוספת הקוראת לשיטה **StopFolding**, אולם אינה מקבלת פרמטרים בעצמה.

```
void FinishFolding()
{
   folder.StopFolding(0); // Shutdown immediately
}
```

הערה

השיטה **FinishFolding** היא דוגמה קלאסית ל**מתאם** (Adapter), שיטה הממירה (או מתאמת) את החתימה של שיטה. זהו דפוס שכיח למדי אשר כלול בקבוצת דפוסי הפעולה שבספר **Design Patterns**: **Elements of Reusable Object-Oriented Architecture** מאת Gamma, Helm, Johnson ו-Vlissides (Addison-Wesley Professional, 1994).

במקרים רבים, שיטות תיאום כדוגמת זו שהצגנו הן קטנות מאוד וקל לא להבחין בהן בין שאר השיטות, במיוחד במחלקות גדולות. כמו כן, אין זה סביר שהשיטה תשמש לביצוע פעולות אחרות חוץ מהתאמת השיטה **StopFolding** לנציג. על כן, שפת #C מאפשרת לכתוב שיטות אנונימיות לפתרון בעיות מסוג זה.

שימוש בשיטות אנונימיות בתור מתאם

שיטה אנונימית (Anonymous Method) היא שיטה ללא שם. ייתכן שזה נשמע מוזר, אולם לפניך שיטות שימושיות וחשובות ביותר.

ייתכנו מצבים שבהם יהיה לך בלוק קוד אשר לעולם לא תקרא לו באופן ישיר, אך תרצה אפשרות להפעילו באמצעות נציג. השיטה **FinishFolding** היא דוגמה למקרה שכזה, ומטרתה היחידה היא לפעול כמתאם עבור השיטה **StopFolding**. מציאת שם לשיטה הינה טרחה מסוימת למתכנת, אך כאשר יש מספר רב של שיטות תיאום כאלו ביישום - הן מתחילות להוות עומס של ממש ולא רק טרחה כלשהי. בשיטה אנונימית ניתן להשתמש בכל מקום שניתן להשתמש בו בנציג. עליך רק לכתוב את הקוד בין צמד סוגריים מסולסלים שלפניהן מילת המפתח **delegate**. ניתן להשתמש בשיטה אנונימית כמתאם לשיטה **StopFolding** ולהוסיפה לנציג **stopMachinery** באופן הבא:

```
this.stopMachinery += delegate { folder.StopFolding(0); };
```

כעת אינך צריך ליצור את השיטה **StopFolding**.

תוכל גם להעביר שיטה אנונימית בתור פרמטר במקום נציג:

```
control.Add(delegate { folder.StopFolding(0); } );
```

מאפייני השיטה האנונימית

לשיטות אנונימיות יש מספר תכונות מיוחדות שעליך להיות מודע אליהן, וביניהן התכונות:

- אם יש צורך בפרמטרים, צריך לציינם בין הסוגריים המסולסלים שאחרי מילת המפתח delegate. לדוגמה:

```
control.Add(delegate(int param1, string param2) { /* code that
uses param1 and param2 */ ... });
```

- שיטות אנונימיות יכולות להחזיר ערכים, אולם סוג הערך המוחזר חייב להיות זהה לסוג הערך המוחזר של הנציג שאליו הן מתווספות.
- ניתן להשתמש באופרטור =- להסרת שיטה ממשתנה נציג. אולם אין הגיון בביצוע פעולה כזו מכיוון שכאן היא לא תסיר דבר. שיטות אנונימיות אשר במקרה מכילות את אותו הקוד, הן למעשה מופעים שונים של הקוד.
- הקוד בשיטה אנונימית הוא קוד C# רגיל. ניתן להרכיבו ממשפטים, קריאות לשיטות, הגדרות משתנים וכו'.
- משתנים שהוגדרו במסגרת שיטה אנונימית יוצאים מתחום ההכרזה כאשר השיטה מסתיימת.
- גישה אנונימית יכולה לגשת למשתנים ולשנות אם הם נמצאים בתחום ההכרזה כאשר השיטה האנונימית מוגדרת. משתנים ששונו בדרך זו נקראים captured outer variables, ויש להיזהר מהם!

אירועים

בסעיף קודם למדת להכריז על סוג נציג, כיצד לקרוא לנציג וכיצד ליצור מופעים של נציג. אולם, פעולות אלו אינן מספיקות. למרות שהנציגים מאפשרים להפעיל מספר שיטות באופן עקיף, אתה עדין צריך להפעיל את הנציג באופן מפורש. בחלק גדול מהמקרים כדאי ליצור מנגנון שיפעיל את הנציג באופן אוטומטי כאשר מתרחש **אירוע** (event) חשוב. לדוגמה, במפעל האוטומטי ייתכן שיהיה צורך להפעיל את הנציג **stopMachinery** כדי לעצור את כל הציוד בבת-אחת אם אחת המכונות מתחממת יתר על המידה או כושלת. ב-NET Framework, **האירועים** (events) מאפשרים להגדיר וללכוד פעולות מסוימות, ולהפעיל נציג אשר יטפל באירוע. מספר רב של מחלקות ב-NET Framework חושפות (expose) אירועים. מרבית הפקדים שניתן למקם על טופס Windows, וגם הטופס עצמו, משתמשים באירועים כדי לאפשר להפעיל קוד. הקוד יופעל כאשר, לדוגמה, המשתמש ילחץ על אחד הלחצנים או יקליד משהו באחד מהשדות. כמו כן, אתה יכול להגדיר אירועים בעצמך.

הכרזה על אירוע

עליך להגדיר אירוע במחלקות שאמורות לשמש כמקור לאירוע. **מקור לאירוע** (event source) הוא לרוב מחלקה אשר מנטרת את הסביבה שלה **ומציפה אירוע** (raises an event) כאשר מתרחש משהו בעל משמעות. במפעל האוטומטי שלנו, מקור לאירוע יכול להיות מחלקה המנטרת את הטמפרטורה של כל אחת מהמכונות. מחלקת ניטור הטמפרטורה מציפה אירוע "התחממות יתר של מכונה" כאשר אחת המכונות חורגת מסף הקרינה התרמית התקני שלה

(כלומר, היא חמה מדי). האירוע מכיל רשימת שיטות שיש לקרוא להן כאשר יש הצפה, ואשר נקראות לעיתים בשם **מנויים** (subscribers). שיטות אלו צריכות לטפל באירוע התחממות היתר ולנקוט בפעולה מתקנת: במקרה שלנו, כיבוי המכונות.

הכרזה על אירוע מתבצעת באופן דומה להכרזה על שדה. אולם מכיוון שהשימוש באירועים נעשה עם נציגים, אירוע חייב להיות מסוג נציג, ועליך להוסיף לו את מילת המפתח **event**, לפני ההכרזה עליו. נשתמש בנציג **StopMachineryDelegate** כדי להדגים כיצד הדבר מתבצע. העברנו את הנציג למחלקה חדשה בשם **TemperatureMonitor**, המשמשת כממשק עם החיישנים האלקטרונים הבודקים את הטמפרטורה של הציוד (אין ספק שמחלקה זו, ולא המחלקה **Controller**, היא המקום ההגיוני ביותר למיקום הנציג).

```
class TemperatureMonitor
{
   public delegate void StopMachineryDelegate();
   ...
}
```

ניתן להגדיר את האירוע **MachineOverheating**, אשר יפעיל את הנציג **StopMachineryDelegate**:

```
class TemperatureMonitor
{
   public delegate void StopMachineryDelegate();
   public event StopMachineryDelegate MachineOverheating;
   ...
}
```

הלוגיקה במחלקה **TemperatureMonitor** (שאינה מוצגת כאן) מציפה בעת הצורך את האירוע **MachineOverheating** באופן אוטומטי. לכל אירוע יש מקבץ נציגים פנימי משלו, ולכן אין צורך להפעיל את משתנה הנציג באופן ידני.

מינוי לאירוע

בדומה לנציגים, גם האירוע מכיל אופרטור =+ מובנה. כדי להיות מנוי לאירוע עליך להשתמש באופרטור זה. במפעל האוטומטי, התוכנה אשר שולטת בכל אחת מהמכונות יכולה לארגן את הקריאה לשיטות הכיבוי של המכונות בעת **הצפה** (raised) של האירוע **MachineOverheating**. הנה כך:

```
TemperatureMonitor tempMonitor = new TemperatureMonitor();
...
tempMonitor.MachineOverheating += delegate { folder.StopFolding(0) };
tempMonitor.MachineOverheating += welder.FinishWelding;
tempMonitor.MachineOverheating += painter.PaintOff;
```

שים לב שהתחביר זהה לתחביר המשמש להוספת שיטה לנציג. ניתן אפילו להיות מנוי (subscribe) באמצעות שיטות אנונימיות. כאשר **tempMonitor.MachineOverheating** מופעל, הוא קורא לכל השיטות שיש להן מנוי ומכבה את המכונות.

ביטול מינוי לאירוע

מכיוון שלמדת שניתן להשתמש באופרטור =+ כדי לשייך נציג לאירוע, בוודאי כבר ניחשת שהאופרטור =− משמש לניתוק נציג מאירוע. הפעלת האופרטור =− מסירה את השיטה מאוסף הנציגים הפנימי של האירוע. פעולה זו נקראת לעיתים קרובות **ביטול המנוי לאירוע** (unsubscribing from the Event).

הצפת אירוע

בדומה לנציג, ניתן לעורר אירוע באמצעות קריאה לו. כאשר אתה **מעורר אירוע**, או **מציף אירוע** (raise an event), כל הנציגים הקשורים אליו מופעלים בזה אחר זה. לדוגמה, להלן המחלקה **TemperatureMonitor** המכילה שיטה **Notify** פרטית המציפה את האירוע **MachineryOverheating**:

```
class TemperatureMonitor
{
   public delegate void StopMachinerDelegate;
   public event StopMachineryDelegate MachineOverheating;
   ...
   private void Notify()
   {
      if (this.MachineOverheating != null)
      {
         this.MachineOverheating();
      }
   }
   ...
}
```

זהו סגנון שכיח. הבדיקה לערך **null** הכרחית, מכיוון ששדה האירוע נקבע מראש כ-**null** ומפסיק להיות כזה רק כאשר מצורפת אליו שיטה באמצעות האופרטור =+. ניסיון להצפת אירוע null יגרור אחריו חריג **NullReferenceException**. אם הנציג אשר מגדיר את האירוע אמור לקבל פרמטרים, יש להעביר את הארגומנטים המתאימים בעת הצפת האירוע. דוגמאות בנושא זה יופיעו בהמשך הפרק.

חשוב

אירועים מכילים מאפיין בטיחות מובנה שימושי ביותר. ניתן להציף אירוע ציבורי (כגון **MachineOverheating**) רק באמצעות שיטות ששייכות למחלקה המגדירה את האירוע (המחלקה **TemperatureMonitor**). כל ניסיון להציף את האירוע מחוץ למחלקה, יגרום לשגיאה בשלב ההידור.

הבנת אירועי הממשק הגרפי

המחלקות והפקדים של NET Framework המשמשים לבניית **ממשקי משתמש גרפיים** (GUIs) מפעילים אירועים. במחצית השנייה של ספר זה תלמד להשתמש באירועי GUI רבים.

כעת נתבונן בדוגמה שבה המחלקה **Button** נגזרת מהמחלקה **Control**, ולכן יורשת אירוע ציבורי מסוג **EventHandler** בשם **Click**. הבה נראה זאת בקוד. הנציג **EventHandler** אמור לקבל שני פרמטרים: הפניה לאובייקט אשר גרם להצפת האירוע, ואובייקט מסוג **EventArgs** המכיל מידע נוסף לגבי האירוע:

```
namespace System
{
   public delegate void EventHandler(object sender, EventArgs args) ;
   public class EventArgs
   {
      ...
   }
}
namespace System.Windows.Forms
{
   public class Control :
   {
      public event EventHandler Click;
      ...
   }
   public class Button : Control
   {
      ...
   }
}
```

המחלקה **Button** מציפה את האירוע **Click** באופן אוטומטי כאשר המשתמש לוחץ על הלחצן שעל המסך (לא נרחיב כאן בהסבר כיצד זה מתבצע). הסידור הזה מקל על יצירת נציג לשיטה הרצויה ושיוך שלו לאירוע הרלוונטי.

הדוגמה הבאה כוללת טופס Windows שיש בו לחצן בשם **okay**, שיטה בשם **okay_Click** וקוד המקשר בין האירוע **Click** שבלחצן **okay** לבין השיטה **okay_Click**:

```
class Example : System.Windows.Forms.Form
{
   private System.Windows.Forms.Button okay;
   ...
   public Example()
   {
      this.okay = new System.Windows.Forms.Button();
      this.okay.Click += new System.EventHandler(this.okay_Click);
      ...
   }
   private void okay_Click(object sender, System.EventsArgs args)
   {
      // Your code to handle the Click event
   }
}
```

כאשר אתה משתמש בחלון DesignerView בסביבת הפיתוח Visual Studio 2005, ה-IDE מחולל קוד אשר משייך שיטות לאירועים באופן אוטומטי. כל שנותר לך לעשות זה להשלים את הלוגיקה בשיטה המטפלת באירוע.

הערה

ניתן להוסיף שיטה לאירוע מבלי ליצור מופע של הנציג. ראה את המשפט הבא:

```
this.okay.Click += new System.EventHandler(this.okay_Click);
```

שניתן להחליף אותו במשפט הזה:

```
this.okay.Click += this.okay_Click;
```

זכור שרכיב העיצוב של Windows Forms בסביבת הפיתוח מחולל תמיד את הגרסה הראשונה.

האירועים שיוצרות מחלקות GUI נוהגים תמיד על פי אותו דפוס פעולה. האירועים הם מסוג נציג, אשר מקבלים שני ארגומנטים ומחזירים ערך **void**. הארגומנט הראשון לעולם יהיה שולח האירוע והארגומנט השני לעולם יהיה מסוג **EventArgs** (או מחלקה הנגזרת מ-**EventArgs**).

הארגומנט **השולח** (**sender**) מאפשר שימוש חוזר בשיטה עבור מספר אירועים. השיטה המיוצגת יכולה לבדוק את הארגומנט **השולח** ולהגיב בהתאם. לדוגמה, ניתן למנות את אותה שיטה לאירוע **Click** של שני לחצנים שונים (עליך רק להוסיף את השיטה לשני אירועים שונים). כאשר האירוע מוצף, הקוד שבשיטה יכול לבדוק את הארגומנט השולח כדי לדעת איזה מבין שני הלחצנים נלחץ.

שימוש באירועים

בתרגיל הבא, עליך להשתמש באירועים כדי לפשט את התוכנית אשר כתבת בתרגיל הראשון. עליך להוסיף שדה אירוע למחלקה **Ticker** ולמחוק את השיטות **Add** ו-**Remove** המופיעות בה. לאחר מכן ערוך את השיטות **Clock.Start** ו-**Clock.Stop** כדי שיהיו מנויות לאירוע. בנוסף עליך להכיר את האובייקט Timer, המשמש את המחלקה **Ticker** כדי לקבל פעימת שעון בכל שנייה.

שכתוב היישום של השעון הדיגיטלי

1. חזור לסביבת הפיתוח שבה מוצג הפרויקט Delegates. הצג את קובץ המקור Ticker.cs בחלון Code and Text Editor.

במחלקה Ticker שבקובץ זה תמצא את ההכרזה על הנציג Tick:

```
public delegate void Tick(int hh, int mm, int ss);
```

2. הוסף למחלקה Ticker אירוע ציבורי tick מסוג Tick:

המחלקה **Ticker** נראית כעת כך:

```
class Ticker
{
   public delegate void Tick(int hh, int mm, int ss);
   public event Tick tick;
   ...
}
```

3. הפוך את משתנה הנציג tickers שבסוף המחלקה **Ticker** להערה, מכיוון שאין לנו עוד צורך בו:

```
// private Tick tickers;
```

4. הפוך גם את השיטות **Add** ו-**Remove** שבמחלקה **Ticker** להערות.
פעולת ההוספה וההסרה מבוצעת באופן אוטומטי על ידי האופרטורים += ו- -= של האירוע.

5. בחלון Code and Text Editor אתר את השיטה **Ticker.Notify**. תפקיד שיטה זו הוא להפעיל מופע של הנציג **Tick**. אין זה דרוש לנו עוד ולכן ערוך אותה כדי שתקרא לאירוע **tick**. אל תשכח לבצע בדיקה האם **tick** הוא **null** או לא, לפני הקריאה לאירוע.
השיטה **Notify** נראית כעת כך:

```
class Ticker
{
   ...
   private void Notify(int hours, int minutes, int seconds)
   {
      if (this.tick != null)
      {
         this.tick(hours, minutes, seconds);
      }
   }
   ...
}
```

שים לב שהנציג **Tick** כולל פרמטרים, ולכן המשפט אשר מציף את האירוע **tick** חייב להעביר ארגומנט לכל אחד מהם.

6. בחן את המשתנה ticking שנמצא בסוף המחלקה:

```
private System.Timers.Timer ticking = new System.Timers.Timer();
```

המחלקה **Timer** היא חלק מ-NET Framework וניתן להפעיל אותה כדי להציף אירוע בכל פרק זמן נתון. בחן את הבנאי של המחלקה **Ticker**:

```
public Ticker()
{
   this.ticking.Elapsed += new ElapsedEventHandler(this.OnTimedEvent);
   this.ticking.Interval = 1000; // 1 second
   this.Enabled = true;
}
```

המחלקה Timer חושפת את האירוע **Elapsed**, אשר ניתן להציפו בכל פרק זמן נתון על פי המאפיין **Interval**. אם נציב ב-**Interval** את הערך 1000, האירוע **Elapsed** יוצף פעם בשנייה (יחידת המידה לערך זה היא אלפיות שנייה). הטיימר יתחיל כאשר תציב במאפיין **Enable** את הערך **true**. המחלקה **Timer** כוללת גם את הנציג **ElapsedEventHandler**, שתפקידו לציין מהי החתימה הנדרשת משיטות כדי שיהיה ניתן לעשותן מנויות לאירוע **Elapsed**. הבנאי יוצר מופע של הנציג אשר מפנה לשיטה **OnTimedEvent**, שמצידו הינו מנוי לאירוע **Elapsed**. השיטה **OnTimedEvent** שנמצאת במחלקה **Ticker** שואבת מידע לגבי הזמן הנוכחי המועבר לה באמצעות הפרמטר **ElapsedEventArgs**, ולאחר מכן משתמשת במידע זה כדי להציף את האירוע **tick** באמצעות השיטה **Notify**:

```
private void OnTimedEvent(object source, ElapsedEventArgs args)
{
   int hh = args.SignalTime.Hour;
   int mm = args.SignalTime.Minutes;
   int ss = args.SignalTime.Seconds;
   Notify(hh, mm, ss);
}
```

הערה

למידע נוסף על המחלקות **Timer** ו- **ElapsedEventArgs**, עיין ב-Microsoft Visual Studio 2005 Documentation אשר כלול ב- Visual Studio 2005.

7. בחלון Code and Text Editor הצג את קובץ המקור Clock.cs.

8. שנה את השיטה **Clock.Start** כדי שהנציג יחובר לאירוע **tick** של השדה **pulsed** באמצעות האופרטור +=. השיטה **Clock.Start** נראית כעת כך:

```
public void Start()
{
   pulsed.tick += this.RefreshTime;
}
```

9. בחלון Code and Text Editor שנה את השיטה **Clock.Stop** באופן שהנציג ינותק מהאירוע **tick** של השדה **pulsed**, באמצעות האופרטור -=.

השיטה **Clock.Stop** נראית כעת כך:

```
public void Stop()
{
pulsed.tick -= this.RefreshTime;
}
```

10. בתפריט Debug בחר Start Without Debugging. הפרויקט ייבנה ויופעל.

11. לחץ Start.

השעון הדיגיטלי מציג כעת את השעה הנכונה ומתעדכן בכל שנייה.

12. לחץ Stop כדי לוודא שהשעון אכן נעצר.

13. סגור את הטופס.

- **אם ברצונך להמשיך לפרק הבא**

  השאר את Visual Studio 2005 פעילה ועבור לפרק 17.

- **אם ברצונך לכבות כעת את Visual Studio 2005**

  פתח את תפריט Menu ולחץ Exit. אם מופיעה תיבת דו-שיח Save, לחץ Yes.

פרק 16 – טבלה מסכמת

המשימה	צריך
להכריז על סוג נציג.	לכתוב את מילת המפתח delegate, אחריה את סוג הערך המוחזר, אחריו את שם הנציג ולבסוף את סוגי הפרמטרים. לדוגמה: `delegate void Tick();`

להפעיל נציג.	להשתמש בתחביר זהה לזה של הקריאה לשיטה. לדוגמה:
	`Tick m;`
	`...`
	`m();`

ליצור מופע של נציג.	להשתמש בתחביר זהה לתחביר יצירת מופע של מחלקה או מבנה: כתוב את מילת המפתח new, אחריה את שם הסוג (שם הנציג), ואחריו את הארגומנטים בין צמד סוגריים מסולסלים. הארגומנט צריך להיות שיטה אשר חתימתה זהה לחתימת הנציג. לדוגמה:
	`delegate void Tick();`
	`private void Method() { ... }`
	`...`
	`Tick m = new Tick(this.Method);`

להכריז על אירוע.	לכתוב את מילת המפתח event, אחריה את שם הסוג (הסוג חייב להיות מסוג נציג), ולבסוף את שם האירוע. לדוגמה:
	`delegate void TickHandler();`
	`class Ticker`
	`{`
	`  public event TickHandler Tick;`
	`}`

המשימה	צריך
לעשות מנוי (subscribe) לאירוע.	ליצור מופע של נציג (מסוג זהה לזה של האירוע), ולחבר את המופע של הנציג לאירוע באמצעות האופרטור +=. לדוגמה:

```
class Clock
{
   ...
   public void Start()
   {
       ticker.Tick += new TickHandler
       (this.RefreshTime);
   }
   private void RefreshTime()
   {
       ...
   }
   private Ticker ticker = new Ticker();
}
```

ניתן גם לגרום למהדר לחולל את הנציג החדש באופן אוטומטי. כל שצריך לעשות, זה לציין את שם השיטה אותה רוצים למנות:

```
public void Start()
{
    ticker.Tick += this.RefreshTime;
}
```

לבטל את המנוי לאירוע (Unsubscribe).	ליצור מופע של נציג (מאותו סוג של האירוע), ולנתק את המופע של הנציג מהאירוע באמצעות האופרטור =-. לדוגמה:

```
class Clock
{
    ...
    public void Stop()
    {
        ticker.Tick -= new TickHandler
            (this.RefreshTime);
    }
    private void RefreshTime()
    {
        ...
    }
    private Ticker ticker = new Ticker();
}
```

או

```
public void Stop()
{
     ticker.Tick -= this.RefreshTime;
}
```

המשימה	צריך
להציף/לעורר (raise) אירוע.	להשתמש בסוגריים באותו אופן המשמש לקריאה לשיטה. יש להעביר ארגומנטים התואמים לסוג האירוע. אסור לשכוח לבדוק האם האירוע הוא null. לדוגמה:

```
class Ticker
{
    public event TickHandler Tick;
    ...
    private void Notify()
    {
        if (this.Tick != null)
        {
            this.Tick();
        }
    }
    ...
}
```

פרק 17

מבוא לגנריות (Generics)

כאשר תסיים פרק זה, תוכל:

- להגדיר מחלקות מוגנות-סוג (type-safe) באמצעות העיקרון Generics.
- ליצור מופע של מחלקה גנרית (generic class) בהתבסס על סוגים שהוגדרו כפרמטר-סוג (type-parameter).
- לממש ממשק גנרי (generic interface).
- להגדיר שיטה גנרית אשר יש בה מימוש אלגוריתם עצמאי מסוג הנתונים עליהם הוא פועל.

בפרק 8 למדת להשתמש בסוג object כדי להתייחס למופע של כל מחלקה שהיא. ניתן גם להשתמש בסוג object כדי לאחסן ערכים מכל הסוגים, וכן ניתן להגדיר פרמטרים באמצעות הסוג object כאשר צריך להעביר לשיטה ערכים מסוגים שאינם קבועים. בנוסף, מחלקה יכולה להחזיר ערך מסוג לא קבוע אם סוג הערך המוחזר שלה מוגדר כ-object. למרות שהגישה הזו מאוד גמישה, היא מקשה על המתכנת, מכיוון שעליו לזכור את סוג המידע שהשתמשו בו, והיא גם עשויה להוביל לשגיאות בזמן ההרצה אם המתכנת שוגה. בפרק זה נלמד על **גנריות** (Generics) – תכונה שנוספה לשפה C# 2.0 ואשר מאפשרת למנוע טעויות מסוג זה.

הבעיה של השימוש באובייקטים

כדי להבין את נושא הגנריות, צריך לבחון תחילה את סוגי הבעיות שהיא אמורה לפתור, ואשר נובעות מהגדרה באמצעות הסוג **object**.

ניתן להשתמש בסוג **object** ביחס לכל סוג ערך או משתנה. כל סוגי ההפניה (reference type) יורשים אוטומטית, באופן ישיר או עקיף, מהמחלקה **System.Object** שב-NET Framework. ניתן להשתמש במידע זה כדי ליצור מחלקות ושיטות כוללניות ביותר. לדוגמה, חלק גדול מהמחלקות השוכנות במרחב השמות **System.Collections** מנצלות את העובדה הזו כדי לאפשר יצירת כל סוג של אוסף (**collection**). בנוסף, במחלקה **System.Collections.Queue** ניתן ליצור תורים (queues) המאחסנים כמעט כל דבר (על מחלקות אוסף למדת בפרק 10). הקוד הבא ממחיש כיצד ניתן ליצור ולשלוט בתור של אובייקטים מסוג **Circle**:

```
using System.Collections;
...
Queue myQueue = new Queue();
Circle myCircle = new Circle();
myQueue.Enqueue(myCircle);
...
myCircle = (Circle)myQueue.Dequeue();
```

השיטה **Enqueue** מוסיפה אובייקט לתחילת התור, בעוד שהשיטה **Dequeue** מסירה את האובייקט מסוף התור. שיטות אלו מוגדרות באופן הבא:

```
public void Enqueue( object item );
public object Dequeue();
```

מכיוון שהשיטות **Enqueue** ו-**Dequeue** עוסקות באובייקטים, ניתן ליצור תורים של **Circles**, **PhoneBooks**, **Clocks** או כל אחת מהמחלקות האחרות אשר הצגנו בפרקים קודמים של ספר זה. אולם חשוב להבחין, שעליך להשליך (cast) את הערך המוחזר על ידי השיטה **Dequeue** על הסוג המתאים, מכיוון שהמהדר אינו מבצע את ההמרה מהסוג object באופן אוטומטי. אם לא תשליך את הערך המוחזר, תופיע השגיאה "Cannot implicitly convert type 'object' to 'Circle'". ראה לדוגמה את מקטע הקוד הבא:

```
Circle myCircle = new Circle();
myQueue.Enqueue(myCircle);
...
myCircle = (Circle)myQueue.Dequeue(); // Cast is mandatory
```

הצורך לבצע השלכה (cast) מפורשת פוגע קשות בגמישות המתקבלת מהשימוש בסוג **object**. קל מאוד לכתוב קוד בצורה הבאה:

```
Queue myQueue = new Queue();
Circle myCircle = new Circle();
myQueue.Enqueue(myCircle);
...
Clock myClock = (Clock)myQueue.Dequeue();
```

קוד זה אומנם יעבור הידור, אולם הוא אינו תקני ולכן יגרום לזריקת החריג **System.InvalidCastException** בסביבת ההרצה. שגיאה זו נובעת מהניסיון לאחסן הפניה לסוג **Circle** במשתנה מסוג **Clock**, למרות ששני הסוגים אינן תואמים. **שגיאה זו מתגלה רק בסביבת ההרצה**, מכיוון שבשלב ההידור אין מספיק מידע כדי לקבוע שזו שגיאה. ניתן לדעת מהו הסוג האמיתי של האובייקט רק כאשר הוא מוסתר מהתור (dequeue) בסביבת ההרצה.

חיסרון נוסף של השימוש בגישת **האובייקטים** ליצירת מחלקות ושיטות כוללניות הוא, שהמרת **האובייקט** לסוג ערך (value type) ולאחר מכן המרתו חזרה לאובייקט על ידי המהדר, גוזלת משאבי זיכרון נוספים וזמן עיבוד נוסף. קח לדוגמה את מקטע הקוד הבא אשר שולט בתור של שלמים (**int**):

```
Queue myQueue = new Queue();
int myInt = 99;
myQueue.Enqueue(myInt);          // box the int to an object
...
myInt = (int)myQueue.Dequeue();  // unbox the object to an int
```

הסוג **Queue** מכיל פריטים מסוג הפניה. הוספת פריטים מסוג ערך לתור, כגון **int**, מחייבת את אריזתם (box) והמרתם לסוגי ההפניה. ובדומה, גם בעת הסרתם מהתור והעברתם אל **int**, יש לפרוק (unbox) אותם ולהמירם חזרה לסוגי ערך (על אריזה ופריקה ראה פרק 8).

למרות שהאריזה והפריקה מתבצעות באופן שקוף, הן פוגעות בביצועים מכיוון שדרושה להן הקצאת זיכרון דינמית. אומנם הפגיעה אינה גדולה כשבוחנים כל פריט בנפרד, אולם יש הצטברות במקרים בהם התוכנית יוצרת תורים ארוכים של סוגי ערך.

הפתרון הגנרי

עקרון הגנריות (Generics) התווסף ל- C# 2.0 כדי לבטל את הצורך בהשלכה (casting), לשפר את אבטחת הסוג (type safety), להפחית את הצורך באריזה (boxing) ולהקל על יצירה של מחלקות ושיטות גנריות. מחלקות ושיטות גנריות מקבלות **פרמטרים של סוג** (**type parameters**), אשר מציינים את סוג האובייקט שהן פועלות עליו. גרסה 2.0 של NET Framework Class Library כוללת גרסאות גנריות עבור חלק גדול ממחלקות האוסף וממשקי האוסף שבמרחב השמות **System.Collections.Generic**. מקטע הקוד שלהלן מדגים את השימוש במחלקה הגנרית **Queue** שנמצאת במרחב שמות זה, ליצירת תור של אובייקטים מסוג **Circle**:

```
using System.Collections.Generic;
...
Queue<Circle> myQueue = new Queue<Circle>();
Circle myCircle = new Circle();
myQueue.Enqueue(myCircle);
...
myCircle = myQueue.Dequeue();
```

בקוד זה יש שני דברים חדשים:

- השימוש בפרמטר-סוג (type parameter) בין הסוגריים המשולשים, <Circle>, בעת ההכרזה על המשתנה myQueue.
- היעדר ההשלכה (cast) בעת הפעלת השיטה Dequeue.

פרמטר-סוג מציין את סוג האובייקטים שהתור יכול לקבל. כל ההפניות לשיטות בתור הזה יצפו באופן אוטומטי לקבל את הסוג שצוין בפרמטר ולא **object**, ולכן ההשלכה לסוג **Circle** בעת הפעלת השיטה **Dequeue** אינה נחוצה עוד. המהדר יוודא שהסוגים לא עורבבו בטעות, ויודיע על שגיאה בשלב ההידור, ולא בסביבת ההרצה, בכל ניסיון לפרוק פריט מהתור **circleQueue** לאובייקט מסוג **Clock**, לדוגמה.

אם תקרא את התיאור של המחלקה הגנרית **Queue** בתיעוד של סביבת הפיתוח, תראה שהיא מוגדרת בצורה הבאה:

```
public class Queue<T> : …
```

ה-**T** מזהה את הפרמטר-סוג ופועל בתור שומר מקום עבור הסוג האמיתי בזמן ההידור. כאשר אתה כותב קוד שנועד ליצור מופע של המחלקה הגנרית **Queue**, עליך להעביר סוג שיחליף את **T**. בדוגמה האחרונה למשל, הסוג **Circle** ביצע את התפקיד הזה. אם תעיין בשיטות שבמחלקה **Queue<T>** תראה שחלק מהן, כגון **Enqueue** ו-**Dequeue**, מציינות את **T** בתור סוג הפרמטר או סוג הערך מוחזר:

```
public void Enqueue( T item );
public T Dequeue();
```

הפרמטר-סוג **T**, מוחלף עם הסוג שתציין בעת ההכרזה על התור. מעבר לזה, כעת יש למהדר מספיק מידע כדי לבצע בדיקת סוגים קפדנית בעת בניית היישום וכך ללכוד שגיאות של חוסר התאמת סוגים בשלב מוקדם.

עליך להיות מודע לעובדה שההחלפה בין **T** לבין סוג הערך הרצוי לך אינה מנגנון החלפה טקסטואלי גרידא. אלא המהדר מבצע החלפה סמאנטית מורכבת המאפשרת לבחור בכל סוג חוקי במקום **T**. הנה מספר דוגמאות:

```
struct Person
{
...
}
...
Queue<int> intQueue = new Queue<int>();
Queue<Person> personQueue = new Queue<Person>();
Queue<Queue<int>> queueQueue = new Queue<Queue<int>>();
```

שתי הדוגמאות הראשונות יוצרות תורים, בעוד שהדוגמה השלישית יוצרת תור של תורים (של int). במקרה של המשתנה **intQueue**, לדוגמה, המהדר יחולל את הגרסאות הבאות של השיטות **Enqueue** ו-**Dequeue**:

```
public void Enqueue( int item );
public int Dequeue();
```

השווה בין הגדרות אלו להגדרות של המחלקה **Queue** שאינה גנרית, שהוצגו קודם. בשיטות אלו, אשר נגזרו מהמחלקה הגנרית, הפרמטר **item** המועבר ל-**Enqueue** הוא מסוג ערך ואינו מחייב אריזה. כמו כן, הערך המוחזר על ידי השיטה **Dequeue** גם הוא מסוג ערך ואינו מחייב פריקה.

מחלקות גנריות יכולות לקבל יותר מפרמטר-ערך אחד. לדוגמה, המחלקה הגנרית **System.Collections.Generic.Dictionary** אמורה לקבל שני פרמטרים של ערך: אחד עבור המפתחות, ואחד עבור הערכים. הדוגמה הבאה ממחישה כיצד לציין מספר פרמטרים של ערך:

```
public class Dictionary<T, U>
```

במילון יש אוסף של צמדי מפתח-ערך. את הערכים (מסוג **U**) יש לאחסן לצד המפתחות (מסוג **T**) השייכים להם, וכדי למשוך אותם צריך לציין את המפתח אשר תחתיו הם מאוחסנים. המחלקה **Dictionary** מכילה סדרן אשר מאפשר לגשת לפריטים כמו במערך. המחלקה מוגדרת כך:

```
public virtual U this[ T key ] { get; set; }
```

שים לב שהסדרן ניגש לערכים מסוג **U** באמצעות מפתח מסוג **T**. כדי ליצור ולהשתמש במילון בשם **directory** המכיל ערכים מסוג **Person** אשר ממוינים באמצעות מפתחות מסוג **string**, ניתן להשתמש בקוד הבא:

```
struct Person
{
...
}
...
Dictionary<string, Person> directory = new Dictionary<string, Person>();
Person john = new Person();
directory["John"] = john;
...
john = directory["John"];
```

בדומה למקרה של המחלקה הגנרית **Queue**, המהדר יזהה ניסיונות לאחסן ערכים שאינם מסוג המבנה **Person**, וגם יוודא שהמפתחות יהיו תמיד מסוג **string**. מידע נוסף על המחלקה **Dictionary** ניתן למצוא בתיעוד של סביבת הפיתוח.

הערה

התחביר המשמש להגדרת מחלקות גנריות יכול לשמש גם להגדרת מבנים (structs) וממשקים גנריים.

מחלקות גנריות לעומת כוללניות

חשוב להבין שמחלקות גנריות (generic) המשתמשות בפרמטרים של סוג (type parameters) שונות ממחלקות גנריות (**generalized**) אשר אמורות לקבל פרמטרים שניתן להשליכם (cast) על סוגים אחרים. לדוגמה, המחלקה **System.Collection.Queue** היא מחלקה גנרית. קיים מימוש אחד בלבד של המחלקה, והשיטות שלה מקבלות פרמטרים מסוג **object** ומחזירות סוג **object**. ניתן להשתמש במחלקה זו עם **int**, **string** וסוגים רבים נוספים, אולם כולם מהווים מופעים של אותה המחלקה.

השווה מחלקה זו עם המחלקה **System.Collection.Generic.Queue<T>**. בכל פעם שאתה משתמש במחלקה זו עם פרמטר-סוג אחר (כגון **Queue<int>** או **Queue<string>**) אתה למעשה גורם למהדר לחולל מחלקה חדשה לגמרי, אשר יכולה לבצע פעולות שהוגדרו על ידי המחלקה הגנרית. ניתן לראות במחלקה הגנרית מחלקה אשר מגדירה תבנית המשמשת את המהדר ליצירת מחלקה עבור סוג מסוים על פי דרישה. הגרסאות השונות של המחלקה הגנרית (**Queue<int>**, **Queue<string>** וכו') נקראות **סוגים בנויים** (**constructed types**).

גנריות ואילוצים

לעיתים יש מצבים שרוצים להבטיח בהם שהפרמטר-סוג המשמש את המחלקה הגנרית יוכל לזהות סוג המכיל שיטות מסוימות. לדוגמה, בעת הגדרה של מחלקה **PrintableCollection** (אוסף של פריטים הניתנים להדפסה), ייתכן שתרצה להבטיח שכל האובייקטים המאוחסנים בה כוללים את השיטה **Print**. ניתן לפקח על תנאי זה באמצעות **אילוץ** (**constraint**).

שימוש באילוץ מאפשר להגביל את הפרמטר-סוג של מחלקה גנרית לסוגים המממשים קבוצה מסוימת של ממשקים, ולפיכך מספקים את השיטות שהוגדרו על ידי הממשקים האלה. לדוגמה, אם הממשק **IPrintable** היה מכיל את השיטה **Print**, היית יכול להגדיר את המחלקה **PrintableCollection** באופן הבא:

```
public class PrintableCollection<T> where T : IPrintable
```

בזמן הידור המחלקה, המהדר בודק כדי לוודא שהסוג המחליף את **T** באמת ממש את הממשק **IPrintable**. הוא יכריז על שגיאה בשלב ההידור אם תנאי זה לא מתקיים.

יצירת מחלקות גנריות

NET Framework Class Library מכילה מספר מחלקות גנריות שאתה יכול להשתמש בהן. בחלק זה תלמד להגדיר מחלקות גנריות משלך, אך לפני שנתחיל נציג בפניך רקע תיאורטי לנושא.

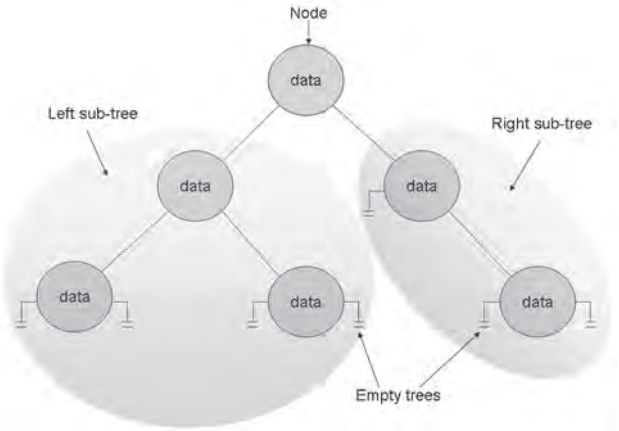
תיאוריית העצים הבינאריים

בתרגילים הבאים עליך להגדיר מחלקות המייצגות עצים בינאריים ולהשתמש בהן. זהו תרגיל שימושי, מכיוון שמחלקה זו איננה נמצאת במרחב השמות **System.Collection.Generic**. עץ בינארי הוא מבנה נתונים חשוב, המשמש לביצוע מגוון פעולות וביניהן, מיון מהיר וחיפוש מהיר של נתונים. יש ספרים רבים העוסקים בעצים בינאריים, אולם איננו רוצים להרחיב כאן את הדיון אלא רק להציג את הפרטים הרלוונטיים לצורך העניין. מידע נוסף בנושא תוכל למצוא בספרים כדוגמת **The Art of Computer Programming, Volume 3: Sorting and Searching** מאת **Donald E. Knuth** (הוצאת Addison-Wesley Professional, 1998).

עץ בינארי הוא מבנה נתונים רקורסיבי (כזה המפנה לעצמו – recursive), אשר יכול להיות או ריק או מורכב משלושה מרכיבים: פיסת מידע המכונה בדרך כלל **צומת** (node) ו**שני ענפים** (sub-trees) אשר כל אחד מהם מהווה עץ בינארי בפני עצמו. שני הענפים נקראים **ענף ימני** (**right sub-tree**) ו**ענף שמאלי** (**left sub-tree**), מכיוון שהם נמצאים מצידו הימני ומצידו השמאלי של הצומת בהתאמה. ענפים אלה יכולים להיות ריקים, או להכיל צומת (node) ושני ענפים משלהם. מבחינה תיאורתית, מבנה זה יכול להימשך עד אינסוף. תרשים 1-17 ממחיש את המבנה של עץ בינארי קטן.

עוצמתם האמיתית של עצים בינאריים באה לידי ביטוי כאשר משתמשים בהם למיון של נתונים. אם נקודת המוצא שלך היא רצף שרירותי של אובייקטים מאותו הסוג, תוכל להשתמש ברצף זה כדי לבנות עץ בינארי מסודר, ולאחר מכן לסרוק את העץ ולבקר בכל אחד מהצמתים (nodes) בסדר קבוע. זהו האלגוריתם המשמש להוספת מרכיב לעץ בינארי מסודר.

```
If the tree, T, is empty
Then
   Construct a new tree T with the new item I as the node, and
     empty left and right sub-trees
Else
   Examine the value of the node, N, of the tree, T
   If the value of N is greater than that of the new item, I
   Then
      If the left sub-tree of T is empty
      Then
         Construct a new left sub-tree of T with the item I as the
           node, and empty left and right sub-trees
      Else
         Insert I into the left sub-tree of T
      End If
   Else
      If the right sub-tree of T is empty
      Then
         Construct a new right sub-tree of T with the item I as the
           node, and
         empty left and right sub-trees
      Else
         Insert I into the right sub-tree of T
      End If
   End If
End If
```



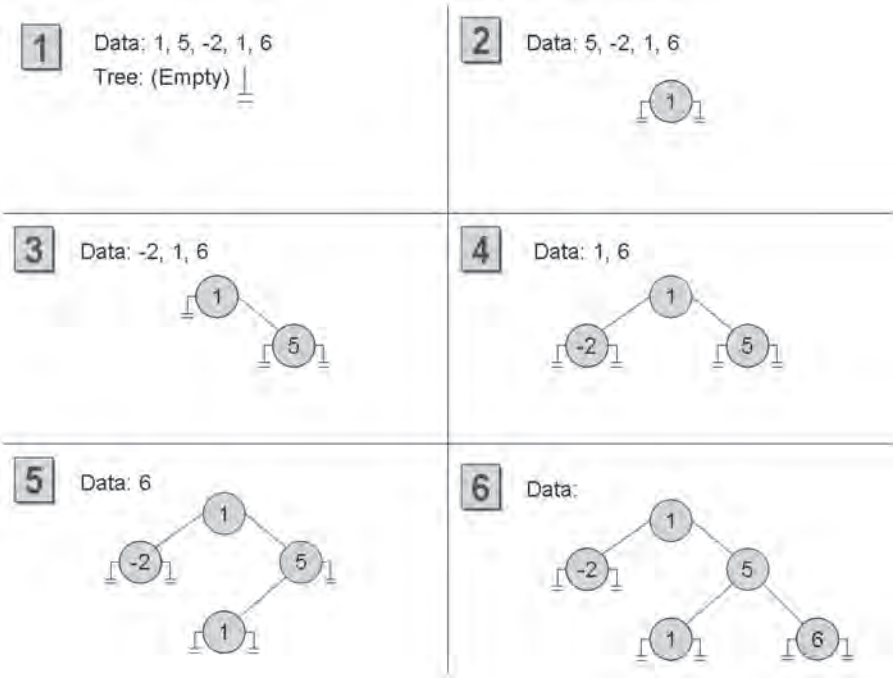
תרשים 17-1: מבנה עץ בינארי

שים לב שאלגוריתם זה הוא רקורסיבי, אשר קורא לעצמו כדי להוסיף את הפריט לענף השמאלי או הימני בהתאם לערך של הפריט והערך של הצומת הנוכחי בעץ.

הערה

הגדרת הביטוי **greater than** מותנית בסוג הנתון הנמצא בפריט ובצומת. כאשר עוסקים בנתונים נומריים, greater than עשוי להיות השוואה אריתמטית פשוטה. כאשר עוסקים בנתוני טקסט ניתן להשתמש בהשוואת מחרוזות. אולם סוגי נתונים אחרים מחייבים אמצעים ייחודיים כדי להשוות בין ערכיהם. נדון על כך בהרחבה כאשר נממש את העץ הבינארי בהמשך הפרק.

אם נקודת המוצא שלך היא עץ בינארי ריק ורצף שרירותי של אובייקטים, תוכל לדפדף בין הפריטים שברצף ולצרף כל אחד מהם לעץ הבינארי באמצעות אלגוריתם זה כדי לקבל בסופו של דבר עץ מסודר. תרשים 17-2 ממחיש את השלבים השונים בתהליך בניית עץ מקבוצה של שלושה מספרים שלמים.

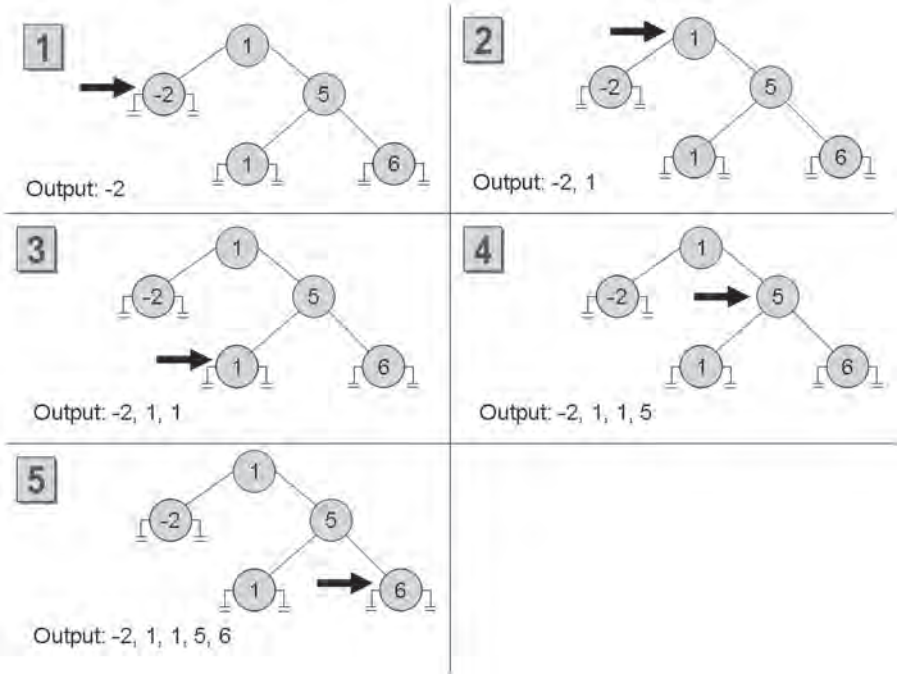


תרשים 17-2: שלבים שונים בתהליך בניית עץ בינארי

לאחר בניית העץ תוכל להציג את תוכנו ברצף על ידי ביקור בכל אחד מהצמתים והדפסת הערך המאוחסן בהם. האלגוריתם לביצוע פעולה זו גם כן רקורסיבי:

```
If the left sub-tree is not empty
Then
   Display the contents of the left sub-tree
End If
Display the value of the node
If the right sub-tree is not empty
Then
   Display the contents of the right sub-tree
End If
```

תרשים 17-3 ממחיש את השלבים בהדפסת תוכן העץ שהרכבנו קודם לכן. שים לב שהמספרים השלמים מוצגים כעת בסדר עולה.



תרשים 17-3: שלבים בהדפסת תוכן עץ בינארי

בניית מחלקת עץ בינארי בצורה גנרית

בתרגיל הבא, עליך להשתמש בעיקרון הגנריות כדי להגדיר מחלקת עץ בינארי אשר יכולה לאחסן כמעט כל סוג של נתונים. המגבלה היחידה היא שסוג הנתון חייב לכלול אמצעי המאפשר השוואה בין הערכים של המופעים השונים.

מחלקת העץ הבינארי היא מחלקה אשר תוכל לשמש אותך במספר רב של יישומים שונים. על כן עליך לממש אותה בתור ספריית מחלקה (class library) ולא בתור ממשק נפרד. כך תוכל להשתמש שוב ושוב במחלקה מבלי להעתיק את קוד המקור ולהדר אותו שוב. ספריית מחלקה היא קבוצה של מחלקות מהודרות וסוגים אחרים כגון מבנים (structs) ונציגים (delegates) המאוחסנים באסופה (assembly). אסופה הוא קובץ המסתיים לרוב בסיומת dll. פרויקטים ויישומים אחרים יכולים להשתמש בפריטים שבספריית מחלקה באמצעות הוספת הפניה לאסופה שבה הם נמצאים, ולאחר מכן להחיל את תחום ההכרזה (scope) של מרחבי השמות (namespaces) שלהם באמצעות מילת המפתח **using**. תעשה זאת בהמשך כאשר תנסה את מחלקת העץ הבינארי.

הממשקים System.IComparable ו- System.IComparable<T>

אם ברצונך ליצור מחלקה העוסקת בהשוואה של ערכים על פי סידור טבעי או מאולץ (natural/unnatural ordering), עליך לממש את הממשק **IComparable**. ממשק זה מכיל שיטה בשם **CompareTo**, אשר מקבלת פרמטר יחיד המציין את האובייקט שיש להשוות למופע הנוכחי, ומחזירה מספר שלם המייצג את תוצאת ההשוואה לפי הטבלה הבאה:

הערך	המשמעות
קטן מאפס	המופע הנוכחי קטן מערך הפרמטר.
אפס	המופע הנוכחי שווה לערך הפרמטר.
גדול מאפס	המופע הנוכחי גדול מערך הפרמטר.

קח לדוגמה את המחלקה **Circle** מפרק 7, שמוצגת שוב להלן:

```
class Circle
{
   public Circle(double initialRadius)
   {
      radius = initialRadius;
   }
   public double Area()
   {
      return 3.141593 * radius * radius;
   }
   private double radius;
}
```

ניתן להשוות את המופעים של המחלקה Circle על ידי מימוש הממשק System.Icomparable ובאמצעות השיטה CompareTo המופיעה להלן. בדוגמה הבאה, השיטה CompareTo משווה בין אובייקטים מסוג Circle על פי שטחם: עיגול ששטחו גדול, יותר גדול מעיגול ששטחו קטן.

```
class Circle : System.IComparable
{
   ...
   public int CompareTo(object obj)
   {
      // cast the parameter to its real type
      Circle circObj = (Circle)obj;
      if (this.Area() == circObj.Area())
         return 0;
      if (this.Area() > circObj.Area())
         return 1;
         return -1;
   }
}
```

אם תעיין בממשק **System.IComparable**, תראה שהפרמטר מוגדר כאובייקט. גישה זו אינה בטוחת-סוג (typesafe). כדי להבין מדוע, חשוב לרגע מה יקרה אם תנסה להעביר פרמטר שאינו **Circle** לשיטה **CompareTo**. הממשק **System.IComparable** חייב שתתבצע השלכה (cast) כדי שתתאפשר לו גישה לשיטה **Area**. אם הפרמטר אינו **Circle** אלא סוג אחר של אובייקט, ההשלכה תיכשל. מרחב השמות **System** מגדיר גם את הממשק הגנרי **IComparable<T>**, המכיל את השיטות הבאות:

```
int CompareTo(T other);
bool Equals(T other);
```

שים לב לשיטה הנוספת, **Equals**, אשר מחזירה את הערך **true** אם שני המופעים זהים זה לזה, ומחזירה **false** אם הם אינם שווים.

שים לב גם לכך ששיטות אלו מקבלות פרמטר-סוג (T) ולא אובייקט, ולפיכך הן בטוחות הרבה יותר מהגרסה הלא גנרית של הממשק. הקוד שלהלן מראה כיצד ניתן לממש ממשק זה במחלקה Circle:

```
class Circle : System.IComparable<Circle>
{
   ...
   public int CompareTo(Circle other)
   {
      if (this.Area() == other.Area())
         return 0;
      if (this.Area() > other.Area())
         return 1;
         return -1;
   }
   public bool Equals(Circle other)
   {
      return (this.CompareTo(other) == 0);
   }
}
```

הפרמטרים עבור השיטות **CompareTo** ו-**Equals** חייבים להיות מאותו סוג שצוין בממשק **IComparable<Circle>**. לרוב עדיף לממש את הממשק **System.IComparable<T>** ולא את **System.IComparable**. ניתן גם לממש את שניהם, כפי שעושים סוגים רבים ב-NET Framework.

כתיבת המחלקה Tree<T>

1. הפעל את Microsoft Visual Studio 2005.
2. צור פרויקט Visual C# חדש באמצעות התבנית Class Library (בתפריט File, New בחר Project). קרא לפרויקט **Binary Tree** ואחסן אותו בתיקייה **Microsoft Press\Visual CSharp Step By Step\Chapter 17**.

3. ב- Solution Explorer שנה את שם הקובץ Class1.cs ל- Tree.cs.

4. בחלון Code and Text Editor, שנה את הגדרת המחלקה **Class1** ל- **Tree<T>**. הגדרת המחלקה **Tree<T>** תהיה כעת כך:

```
public class Tree<T>
{
   ...
}
```

הערה

נהוג להשתמש באות אחת (לרוב T) כדי לסמן את הפרמטר-סוג, למרות שניתן להשתמש בכל מזהה תקני של #C. אם מחלקה גנרית משתמשת במספר פרמטרים של סוג, עליך להשתמש במזהה שונה לכל סוג. לדוגמה, המחלקה הגנרית **Dictionary** ב- NET Framework. מוגדרת באופן הבא: **Dictionary<K, V>**. הסוג K הוא סוג המפתחות של המילון, והסוג V הוא הסוג של ערכי המילון.

5. בחלון Code and Text Editor, שנה את הגדרת המחלקה **Tree<T>** כך שהפרמטר-סוג T יהיה חייב לקבל סוג אשר כולל מימוש של הממשק הגנרי **IComparable<T>**.
ההגדרות של **Tree<T>** נראות כעת כך:

```
public class Tree<T> where T : IComparable<T>
{
   ...
}
```

6. הוסף שלושה משתנים פרטים למחלקה **Tree<T>**. משתנה T בשם **data**, ושני משתני **Tree<T>** בשם **left** ו-**right**:

```
private T data;
private Tree<T> left;
private Tree<T> right;
```

7. הוסף בנאי למחלקה **Tree<T>** אשר מקבל פרמטר **T** בודד בשם **nodeValue**. בבנאי, הצב את **nodeValue** במשתנה **data**, ואתחל את המשתנים **left** ו-**right** למצב **null**, כלהלן:

```
public Tree(T nodeValue)
{
   this.data = nodeValue;
   this.left = null;
   this.right = null;
}
```

8. הוסף למחלקה **Tree<T>** מאפיין ציבורי מסוג **T** בשם **NodeData**. המאפיין אמור לכלול אחראי גישה (accessors) מסוג **get** ו-**set** המאפשרים למשתמש לקרוא ולערוך את המשתנה **data**.

המאפיין **NodeData** נראה כעת כך:

```
public T NodeData
{
   get { return this.data; }
   set { this.data = value; }
}
```

9. הוסף שני מאפיינים נוספים למחלקה **Tree<T>** הנקראים **LeftTree** ו-**RightTree**. מאפיינים אלה דומים מאוד זה לזה ושניהם מספקים, בהתאמה, אחראי גישה **get** ו-**set** למשתנים **left** ו-**right** מהסוג **Tree<T>**.

המאפיינים **LeftTree** ו **RightTree** נראים כעת כך:

```
public Tree<T> LeftTree
{
   get { return this.left; }
   set { this.left = value; }
}
public Tree<T> RightTree
{
   get { return this.right; }
   set { this.right = value; }
}
```

10. הוסף שיטה ציבורית בשם **Insert** למחלקה **Tree<T>**. השיטה צריכה להוסיף ערך **T** לעץ.

הגדרת השיטה נראית כעת כך:

```
public void Insert (T newItem)
{
}
```

השיטה **Insert** פועלת על פי האלגוריתם אשר הוצג קודם לכן. המתכנת משתמש בבנאי כדי להוסיף את הצומת הראשון לעץ, ולכן השיטה **Insert** יכולה לפעול מנקודת הנחה שהעץ אינו ריק. החלק השני של האלגוריתם מוצג להלן, כדי להקל עליך את הבנת הקוד שעליך לכתוב בשלבים הבאים:

```
...
Examine the value of the node, N, of the tree, T
If the value of N is greater than that of the new item, I
Then
   If the left sub-tree of T is empty
   Then
      Construct a new left sub-tree of T with the item I as the
         node, and empty
      left and right sub-trees
   Else
      Insert I into the left sub-tree of T End If
...
```

11. בשיטה **Insert**, הוסף משפט אשר מכריז על משתנה מקומי **currentNodeValue** מסוג **T**. אתחל משתנה זה לערך של המאפיין **NodeData** של העץ, כמוצג להלן:

```
public void Insert(T newItem)
{
   T currentNodeValue = this.NodeData;
}
```

12. הוסף את המשפט **if-else** הבא אל השיטה **Insert**, מייד לאחר הגדרת המשתנה **currentNodeValue**. משפט זה משתמש בשיטה **CompareTo** מהממשק **IComparable<T>** כדי לקבוע אם הערך של הצומת הנוכחי גדול מהפריט החדש:

```
if (currentNodeValue.CompareTo(newItem) > 0)
{
   //Insert the new item into the left sub-tree
}
else
{
   // Insert the new item into the right sub-tree
}
```

13. החלף את ההערה //Insert the new item into the left sub-tree עם בלוק הקוד הבא:

```
if (this.LeftTree == null)
{
    this.LeftTree = new Tree<T>(newItem);
}
else
{
    this.LeftTree.Insert(newItem);
}
```

משפטים אלה בודקים אם הענף השמאלי ריק או לא. אם הוא ריק, נוצר ענף שמאלי חדש עבור הפריט החדש. אם לא, הפריט החדש מאוחסן בענף השמאלי הקיים באמצעות קריאה רקורסיבית (recursive) לשיטה **Insert**.

14. החלף את ההערה //Insert the new item into the right sub-tree עם קוד דומה אשר מוסיף את הרכיב החדש לענף הימני:

```
if (this.RightTree == null)
{
    this.RightTree = new Tree<T>(newItem);
}
else
{
    this.RightTree.Insert(newItem);
}
```

15. הוסף למחלקה **Tree<T>** שיטה ציבורית בשם **WalkTree**. שיטה זו עוברת על העץ, מבקרת בכל אחד מהצמתים על פי הסדר, ומדפיסה את ערכם.

הגדרת השיטה נראית כעת כך:

```
public void WalkTree()
{
    if (this.LeftTree != null)
    {
        this.LeftTree.WalkTree();
    }
    Console.WriteLine(this.NodeData.ToString());

    if (this.RightTree != null)
    {
        this.RightTree.WalkTree();
    }
}
```

16. בתפריט Build בחר Build Solution. הידור המחלקה אמור לעבוד כשורה. על כן, אם מופיעות שגיאות, תקן אותן ובנה את התוכנית שוב.

בתרגיל הבא עליך ליצור עצים בינאריים של שלמים ומחרוזות, כדי לנסות את המחלקה Tree<T>.

בדוק את המחלקה Tree<T>

1. בתפריט File, Add בחר New Project. הוסף פרויקט חדש באמצעות התבנית Console Application. קרא לפרויקט **BinaryTreeTest** ואחסן אותו בתיקייה **\Microsoft Press\Visual CSharp Step By Step\Chapter 17**.

הערה

עליך לזכור שפתרון של Visual Studio 2005 יכול להכיל יותר מפרויקט אחד. תכונה זו מאפשרת להוסיף פרויקט נוסף לפתרון BinaryTree כדי לבדוק את המחלקה **Tree<T>**. זוהי הדרך המומלצת לבדיקת ספריות מחלקה.

2. ב- Solution Explorer, ודא שהפרויקט BinaryTreeTest מסומן. בתפריט Project בחר Set as Startup Project. כתוצאה, הפרויקט יודגש בחלון Solution Explorer.

בעת הפעלת היישום, יופעל למעשה הפרויקט.

3. בתפריט Project בחר Add Reference. בתיבת הדו-שיח Add Reference לחץ על הכרטיסייה Projects. לחץ על הפרויקט BinaryTree ואחר כך לחץ OK.

כעת תופיע האסופה (assembly) BinaryTree ברשימת ההפניות של הפרויקט BinaryTreeTest שב- Solution Explorer. כעת ניתן ליצור אובייקטים מסוג **Tree<T>** בפרויקט BinaryTreeTest.

הערה

אם פרויקט ספריית המחלקה אינו חלק מפתרון הפרויקט אשר עושה בו שימוש, עליך להוסיף הפניה לאסופה (לקובץ dll) ולא לפרויקט ספריית המחלקה. כדי לעשות זאת עליך לבחור את האסופה (הקובץ) מהכרטיסייה Browse שבתיבת הדו-שיח Add Reference (קובץ זה ייווצר לאחר הידור הפרויקט). תוכל לתרגל את הטכניקה הזאת בתרגילים שבהמשך הפרק.

4. בחלון Code and Text Editor המציג את המחלקה Program, הוסף את ההנחיה **using** הבאה לרשימה שבראש המחלקה.

```
using BinaryTree;
```

5. הוסף את המשפטים הבאים לשיטה Main.

```
Tree<int> tree1 = new Tree<int>(10);
tree1.Insert(5);
tree1.Insert(11);
tree1.Insert(5);
tree1.Insert(-12);
tree1.Insert(15);
tree1.Insert(0);
tree1.Insert(14);
tree1.Insert(-8);
tree1.Insert(10);
tree1.Insert(8);
tree1.Insert(8);
tree1.WalkTree();
```

משפטים אלו יוצרים עץ בינארי חדש לאחסון מספרים שלמים. הבנאי יוצר צומת ראשוני שמכיל את הערך 10. משפטי **Insert** מוסיפים צמתים לעץ, והשיטה **WalkTree** מדפיסה את תכולת העץ המסודרת בסדר עולה.

הערה

זכור שמילת המפתח int ב-C# הינה למעשה שם קיצור לסוג **System.Int32**. כל הכרזה על סוג **int** היא הכרזה על משתנה **מבנה** (**struct**) מסוג **System.Int32**. הסוג **System.Int32** מממש את הממשקים **IComparable** ו- **IComparable<T>**, ולכן אינך יכול ליצור משתנים מסוג **Tree<T>**. כמו כן, מילת המפתח string היא הקיצור של System.String, הממשש גם את **IComparable** ואת **IComparable<T>**.

6. בתפריט Build בחר Build Solution. ודא שניתן להדר את הפתרון ותקן שגיאות במידת הצורך.

7. בתפריט Debug בחר Start Without Debugging.

כאשר התוכנית פועלת, הערכים אמורים להופיע בסדר הבא (קרא מימין לשמאל):

-12, -8, 0, 5, 5, 8, 8, 10, 10, 11, 14, 15.

8. הוסף את המשפטים הבאים בסוף השיטה Main, לאחר הקוד הקיים:

```
Tree<string> tree2 = new Tree<string>("Hello");
tree2.Insert("World");
tree2.Insert("How");
tree2.Insert("Are");
tree2.Insert("You");
tree2.Insert("Today");
tree2.Insert("I");
tree2.Insert("Hope");
tree2.Insert("You");
tree2.Insert("Are");
tree2.Insert("Feeling");
tree2.Insert("Well");
tree2.Wal
```

משפטים אלה יוצרים עץ בינארי נוסף המכיל מחרוזות, מאחסנים בו נתונים ולבסוף מדפיסים את תכולתו. הפעם המידע יסודר בסדר אלפביתי.

9. בתפריט Build בחר Build Solution. ודא שניתן להדר את הפתרון ותקן שגיאות במידת הצורך.

10. בתפריט Debug בחר Start Without Debugging.

כאשר התוכנית פועלת, הערכים השלמים יופיעו על המסך ואחריהם יופיעו המחרוזות בסדר הבא (קרא מימין לשמאל):

Are, Are, Feeling, Hello, Hope, How, I, Today, Well, World, You, You.

11. הקש Enter כדי לחזור אל Visual Studio 2005.

יצירת שיטה גנרית

בנוסף להגדרת מחלקות גנריות, באפשרותך להשתמש ב- NET Framework גם ליצירה של שיטות גנריות.

שיטה גנרית מאפשרת לציין פרמטרים וסוג ערך מוחזר באמצעות פרמטר-סוג באופן דומה להגדרות המחלקות הגנריות. בדרך זו ניתן להגדיר מחלקות גנריות בטוחות-סוג (type-safe) תוך כדי ביטול הצורך להקצות משאבים להשלכה (casting), ובמקרים מסוימים גם לאריזה (boxing). לעיתים קרובות המחלקות והישויות הגנריות פועלות זו לצד זו ־ מחלקות מסוימות מקבלות סוג מחלקה גנרית כפרמטר או מחזירות ערך מסוג מחלקה גנרית.

התחביר המשמש להגדרת שיטות גנריות כולל את אותם פרמטרים של סוג (type-parameters) המשמשים ליצירת מחלקות גנריות, וניתן גם להוסיף אילוצים (constraints). לדוגמה, בשיטה הגנרית **Swap<T>** המוצגת להלן ניתן להשתמש כדי להחליף בין ערכי הפרמטרים שהיא

מקבלת. מכיוון שפעולה זו עשויה להיות שימושית עבור נתון מכל סוג שהוא, רצוי להגדיר שיטה זו כגנרית:

```
static void Swap<T>( ref T first, ref T second)
{
   T temp = first;
   first = second;
   second = temp;
}
```

כדי להפעיל את השיטה, עליך להעביר את הסוג המתאים כפרמטר-סוג. הדוגמאות הבאות מציגות שימוש בשיטה **Swap<T>** להחלפה בין שני שלמים והחלפה בין שתי **מחרוזות** (**string**):

```
int a = 1, b = 2;
Swap<int>(ref a, ref b);
...
string s1 = “Hello”, s2 = “World”;
Swap<string>(ref s1, ref s2);
```

הערה

בעת שימוש במחלקה גנרית עם סוג-פרמטר כלשהו, המהדר יוצר סוג חדש. כך גם שימוש בשיטה גנרית עם פרמטר-סוג כלשהו גורם למהדר ליצור גרסה שונה של השיטה. השיטה **Swap<int>** שונה מהשיטה **Swap<string>**. המשותף לשתיהן הוא שהן נוצרו מאותה שיטה גנרית, ולכן הן חולקות במאפייני התנהגות דומים, גם אם עבור נתונים מסוגים שונים.

הגדרת שיטה גנרית לבניית עץ בינארי

בתרגיל האחרון למדת כיצד ליצור מחלקה גנרית למימוש עץ בינארי. המחלקה **Tree<T>** כוללת את השיטה **Insert** המשמשת להוספת נתונים לעץ. אם תרצה להוסיף מספר רב של פריטים, תצטרך לחזור מספר רב של פעמים על הקריאה לשיטה **Insert**. בתרגיל הבא עליך להגדיר מחלקה גנרית **BuildTree** המאפשרת ליצור עץ בינארי חדש באמצעות רשימת נתונים. אחר כך תצטרך לבנות עץ המורכב מתווים כדי לבדוק את השיטה.

כתוב את השיטה BuildTree

1. הפעל את Visual Studio 2005 כדי ליצור פרויקט חדש בעזרת התבנית Console Application. בתיבת הדו-שיח New Project קרא לפרויקט **BuildTree** ושמור אותו בתיקייה **\Microsoft Press\Visual CSharp Step By Step\Chapter 17**. ברשימה הנפתחת Solution בחר Create a new Solution.
2. בתפריט Project בחר Add Reference. בתיבת הדו-שיח Add Reference בחר בכרטיסייה Browse. היכנס לתיקייה **\Microsoft Press\Visual CSharp Step By Step\Chapter 17\BinaryTree\bin\Debug**, בחר בקובץ BinaryTree.dll ולחץ OK.

   האסופה BinaryTree תיווסף כעת לרשימת ההפניות המוצגות ב- Solution Explorer.

3. בחלון Code and Text Editor, הוסף לראש הקובץ Program.cs את ההנחיה **using** הבאה:

```
using BinaryTree;
```

זהו מרחב השמות שמכיל את המחלקה **Tree<T>**.

4. הוסף שיטה בשם **BuildTree** למחלקה **Program**. זו שיטה **סטאטית** (**static**) בשם **data** המקבלת מערכים מסוג **params** המשמשים לאחסון מרכיבים מסוג **T**, ומחזירה אובייקטים מסוג **Tree<T>**.

הגדרת השיטה נראית כעת כך:

```
static Tree<T> BuildTree<T>(params T[] data)
{
}
```

הערה

למידע נוסף על מילת המפתח **params**, ראה פרק 11.

5. הסוג **T** אשר משמש לבניית העץ הבינארי חייב לממש את הממשק **IComparable<T>**. שנה את הגדרות השיטה **BuildTree**, והוסף סעיף **where** מתאים.

הגדרת השיטה המעודכנת נראית כך:

```
static Tree<T> BuildTree<T>(params T[] data) where T :
IComparable<T>
{
}
```

6. הוסף את המשפטים שלהלן לשיטה **BuildTree**. משפטים אלה יוצרים מופע חדש של **Tree** באמצעות הפרמטר-סוג המתאים. לאחר מכן יש דפדוף ברשימת **params** תוך כדי הוספת כל אחד מהנתונים לעץ באמצעות השיטה **Insert**.

העץ המוגמר הוא למעשה הערך המוחזר על ידי השיטה:

```
static Tree<T> BuildTree<T>(params T[] data) where T :
IComparable<T>
{
   Tree<T> sortTree = new Tree<T> (data[0]);
   for (int i = 1; i < data.Length; i++)
   {
       sortTree.Insert(data[i]);
   }
   return sortTree;
}
```

בדיקת השיטה BuildTree

1. בשיטה **Main** של המחלקה **Program**, הוסף את המשפטים הבאים, שתפקידם ליצור עץ לאחסון תווים, לאחסן בו נתונים באמצעות השיטה **BuildTree**, ולבסוף להציג אותם באמצעות השיטה **WalkTree** של העץ:

```
Tree<char> charTree = BuildTree<char>('Z', 'X', 'A', 'M', 'Z', 'M', 'N');
charTree.WalkTree();
```

2. בתפריט Build בחר Build Solution. ודא שניתן להדר את הפתרון ותקן שגיאות במידת הצורך.
3. בתפריט Debug בחר Start Without Debugging.

   כאשר התוכנית עובדת, יופיעו התווים הבאים לפי הסדר (קרא מימין לשמאל):

   A, M, M, N, X, Z, Z.
4. הקש Enter כדי לחזור אל Visual Studio 2005.

☯ **אם ברצונך להמשיך לפרק הבא**

השאר את Visual Studio 2005 פעילה ועבור לפרק 18.

☯ **אם ברצונך לכבות כעת את Visual Studio 2005**

פתח את תפריט Menu ולחץ Exit. אם מופיעה תיבת דו-שיח Save, לחץ Yes.

פרק 17 – טבלה מסכמת

המשימה	צריך	דוגמה
ליצור מופע של סוג גנרי.	לציין את הפרמטר-סוג הרצוי.	`Queue<int> myQueue = new Queue<int>();`
ליצור סוג גנרי חדש.	להגדיר מחלקה באמצעות פרמטר-סוג.	`public class Tree<T> { ... }`
להגביל את הסוג אשר יכול לשמש בתור פרמטר-סוג.	להוסיף אילוץ (constraint) באמצעות סעיף where בעת הגדרת המחלקה.	`public class Tree<T> where T : IComparable<T> { ... }`
להגדיר שיטה גנרית.	להגדיר שיטה באמצעות פרמטר-סוג.	`static Tree<T> BuildTree<T> (params T[] data) { ... }`
להפעיל שיטה גנרית.	לציין סוג ספציפי עבור כל פרמטר-סוג שהשיטה אמורה לקבל.	`Tree<char> charTree = BuildTree<char>('Z', 'X');`

פרק 18

מונים של אוספים

כאשר תסיים פרק זה, תוכל:

- להגדיר באופן ידני **מונה** (enumerator) שיכול לשמש לדפדוף בין מרכיבים של אוסף.
- לממש רשימה באופן אוטומטי באמצעות יצירה של **איטרטור** (iterator, מחזרר).
- ליצור **איטרטורים** נוספים אשר יכולים לעבור בין מרכיבי האוסף בסדר שונה בכל פעם.

בפרק 10 למדת להשתמש במחלקות מערך (array) ומחלקות אוסף (collection) כדי לאחסן רצפים או קבוצות של נתונים. בפרק 10 למדת גם על המשפט **foreach** היכול לשמש כדי לעבור, או לדפדף (iterate), בין מרכיבי האוסף השונים. עד כה השתמשת במשפט **foreach** כדי לגשת לתכולה של אוסף בצורה מהירה ונוחה. כעת תלמד כיצד משפט זה פועל למעשה. חשיבות הנושא באה לידי ביטוי בעת הגדרה של מחלקות אוסף חדשות, במיוחד לאור הוספת איטרטורים (iterators) לשפה C# 2.0, אשר מאפשרים לבצע חלק גדול מהתהליך באופן אוטומטי.

ספירת (Enumerating) מרכיבי האוסף

בפרק 10 ראית דוגמאות לשימוש במשפט **foreach** המציג רכיבים של מערך פשוט. הקוד נראה כך:

```
int[] pins = { 9, 3, 7, 2 };
foreach (int pin in pins)
{
   Console.WriteLine(pin);
}
```

המבנה **foreach** הינו מנגנון אלגנטי המפשט באופן משמעותי את הקוד שעליך לכתוב, אולם ניתן להשתמש בו רק בנסיבות מסוימות - כדי לדפדף באוספים מסוג **enumerable collection** בלבד. אם כך, מהו אוסף מסוג זה? התשובה הקצרה לשאלה זו היא שזהו אוסף למימוש הממשק **System.Collections.IEnumerable**.

הערה

עליך לזכור שכל המערכים שב-C# הינם מופעים של המחלקה **System.Array**. זוהי מחלקת אוסף למימוש הממשק **IEnumerable**.

השיטה **GetEnumerator** מחזירה אובייקט מסוג רשימה (**enumerator**) למימוש הממשק **System.Collections.IEnumerable**.

```
IEnumerator GetEnumerator();
```

אובייקט הרשימה (מונה) משמש למעבר בין רכיבי האוסף בזה אחר זה (enumerating). הממשק IEnumerable עצמו מגדיר את המאפיינים והשיטות הבאות:

```
object Current {get;}
bool MoveNext();
void Reset();
```

חשוב על המונה כמצביע אל קבוצה של אלמנטים ברשימה. בהתחלה ההצבעה היא **לפני** האלמנט הראשון. ניתן לקרוא לשיטה **MoveNext** כדי להעביר את המצביע לפריט הבא. במידה ובאמת יש פריט נוסף, השיטה **MoveNext** צריכה להחזיר ערך **True**. אם אין פריט נוסף, עליה להחזיר **false**. המאפיין **Current** משמש לגישה לפריט שהמצביע מפנה אליו, והשיטה **Reset** משמשת להפנות שוב את המצביע למצב הראשוני לשלו, **לפני** הפריט הראשון. יצירה של מונה באמצעות השיטה **GetEnumerator** של האוסף, קריאה מחזורית לשיטה **MoveNext** ושליפת הערך של המאפיין **Current** באמצעות המונה יאפשרו להתקדם בין האלמנטים השונים של הרשימה בזה אחר זה. זו בדיוק הפעולה שמבצע המשפט **foreach**. לכן, אם ברצונך ליצור מחלקת אוסף מסוג enumerable collections class, עליך לממש את הממשק **IEnumerable** במחלקת האוסף שלך, ועליך גם לכתוב מימוש לממשק **IEnumerable** אשר יהווה את סוג הערך המוחזר על ידי השיטה **GetEnumerator** של מחלקת האוסף.

דקי האבחנה בוודאי ישימו לב שהמאפיין **Current** מהממשק **IEnumerable** מפגין התנהגות **שאינה בטוחת-סוג** (non type-safe), מכיוון שהוא מחזיר סוג object ולא סוג מוגדר יותר. למרבה המזל, NET Framework Class Library כוללת גם ממשק **IEnumerable<T>** גנרי, הכולל מאפיין **Current** אשר מחזיר ערך **T**. כמו כן קיים ממשק **IEnumerable<T>** שמכיל שיטה **GetEnumerator** המחזירה אובייקט מסוג **Enumerator<T>**. אם אתה בונה יישומים עבור NET Framework 2.0, עדיף להשתמש בממשקים הגנריים, ולא בממשקים הרגילים.

הערה

מספר נקודות נוספות מבדילות בין הממשק **IEnumerable<T>** לבין הממשק **IEnumerable**. הראשון אינו מכיל את השיטה **Reset**, אולם הממשק **IDisposable** שלו מפורט יותר.

מימוש ידני של מונה

בתרגיל הבא עליך להגדיר מחלקה למימוש הממשק הגנרי **IEnumerable<T>** וליצירת מונה עבור מחלקת העץ הבינארי שבנית בפרק 17. בפרק 17 למדת עד כמה קל לעבור בין מרכיבי העץ הבינארי ולהציג את תכולתו. אולם למרבה הצער, הגדרת מונה אשר שולף כל מרכיב בעץ הבינארי באותו הסדר אינה פשוטה כלל. הבעיה העיקרית היא שבעת הגדרת מונה עליך לעקוב כל הזמן אחר מיקומך במבנה, כדי שהקריאות הבאות לשיטה **MoveNext** יוכלו

לעדכן את המיקום בצורה הנכונה. באלגוריתמים רקורסיביים, כדוגמת אלה ששימשו לסריקת העץ הבינארי, לא ניתן להשתמש כדי להעביר מידע על המצב הנוכחי בין הקריאות לשיטות בצורה נוחה לגישה. מסיבה זו, עליך לעבד את המידע שבעץ הבינארי למבנה נתונים נוח יותר לשימוש (למשל, על ידי תור, queue) ולמַספר (enumerate) את מבנה הנתונים החדש במקום העץ. כל זה מוסתר מהמשתמש אשר מדפדף במרכיבי העץ הבינארי!

יצירת המחלקה TreeEnumerator

1. הפעל את Microsoft Visual Studio 2005 אם אינה פעילה.
2. פתח את הפתרון **\Microsoft Press\Visual CSharp Step By Step\Chapter 18\BinaryTree\BinaryTree.sln**. פתרון זה כולל עותק של הפרויקט BinaryTree אשר יצרת בפרק 17.
3. הוסף מחלקה חדשה לפרויקט: בתפריט Project בחר Add Class, בחר בתבנית Class, בשדה Name הקלד **TreeEnumerators.cs** ולחץ Add.
4. המחלקה **TreeEnumerator** תחולל מונה עבור האובייקט **Tree<T>**. כדי לוודא שהמחלקה היא typesafe, עליך לכלול פרמטר-סוג ולממש את הממשק **IEnumerator<T>**. בנוסף, הפרמטר-סוג חייב להיות סוג חוקי מבחינת האובייקט **Tree<T>** שהמחלקה מונה, ולכן יש להשתמש באילוץ כדי לחסום סוגים שאינם כוללים את הממשק **IComparable<T>**.

   שנה את הגדרת המחלקה כדי שתעמוד בדרישות הללו. המחלקה נראית כעת כך:

```
public class Tree<T> : IEnumerable<T> where T : IComparable<T>
{
}
```

5. הוסף את שלושת המשתנים הפרטיים הבאים למחלקה TreeEnumerator<T>:

```
private Tree<T> currentData = null;
private T currentItem = default(T);
private Queue<T> enumData = null;
```

המשתנה **currentData** ישמש לאחסון ההפניה לעץ שאנחנו מונים, והמשתנה **currentItem** ישמש לאחסון הערך המוחזר על ידי המאפיין Current. עליך לאחסן את הערכים ששלפת מהצמתים של העץ במשתנה **enumData**, והשיטה **MoveNext** תחזיר בזה אחר זה כל אחד מהפריטים שבתור.

6. הוסף בנאי **TreeEnumerator** אשר מקבל פרמטר יחיד **data** מסוג **Tree<T>**. בגוף הבנאי הוסף משפט המאתחל את המשתנה **currentData** לפרמטר **data**.

```
public TreeEnumerator(Tree<T> data)
{
   this.currentData = data;
}
```

7. הוסף את השיטה הפרטית populate שלהלן אל המחלקה **TreeEnumerator<T>**, מייד לאחר הבנאי:

```
private void populate(Queue<T> enumQueue, Tree<T> tree)
{
   if (tree.LeftTree != null)
   {
       populate(enumQueue, tree.LeftTree);
   }
   enumQueue.Enqueue(tree.NodeData);
   if (tree.RightTree != null)
   {
       populate(enumQueue, tree.RightTree);
   }
}
```

שיטה זו עוברת על עץ בינארי ומוסיפה את הנתונים המאוחסנים בו אל התור. האלגוריתם המשמש אותה דומה מאוד לאלגוריתם שהשתמשנו בו בשיטה **WalkTree** של המחלקה **Tree<T>** (פרק 17). ההבדל העיקרי בין השניים הוא שבמקום שערכי **NodeData** יודפסו על המסך, הם מאוחסנים בתור.

8. חזור להגדרות המחלקה **TreeEnumerator<T>**. לחץ לחיצה ימנית על הממשק **IEnumerator<T>** שבהכרזה על המחלקה, בתפריט הקיצור הצבע על Implement Interface ובחר Implement Interface Explicitly.

פעולה זו מחוללת שלד (תבניות) עבור השיטות השונות של הממשק **IEnumerator<T>** והממשק **IEnumerator**, ומציבה אותן בסוף המחלקה. הפעולה מחוללת גם את השיטה **Dispose** עבור הממשק **IDisposable**.

הערה

הממשק **IEnumerator<T>** יורש מהממשקים **IEnumerator** ו-**IDisposable**, ולכן גם השיטות שלהם נוצרות. למעשה, הפריט היחיד אשר שבאמת שייך ל- **IEnumerator<T>** הוא המאפיין הגנרי **Current**. השיטות **MoveNext** ו-**Reset** שייכות לממשק הלא-גנרי **IEnumerator**. הממשק **IDisposable** מופיע בפרק 13.

9. הבט בקוד שזה עתה נוצר. גופי המאפיינים והשיטות מכילים מימוש ברירת מחדל אשר זורק חריג **Exception** עם ההודעה "The method or operation is not implemented". בצעדים הבאים עליך להחליף את הקוד הזה במימוש אמיתי.

10. החלף את גוף השיטה MoveNext עם הקוד שלהלן:

```
bool System.Collections.IEnumerator.MoveNext()
{
   if (this.enumData == null)
   {
       this.enumData = new Queue<T>();
       populate(this.enumData, this.currentData);
   }
   if (this.enumData.Count > 0)
   {
       this.currentItem = this.enumData.Dequeue();
       return true;
   }
   return false;
}
```

לשיטה **MoveNext** של הרשימה יש שתי מטרות. בפעם הראשונה שקוראים לה היא מאתחלת את הנתונים המשמשים את המונה (enumerator) ומתקדמת אל הנתון הראשון שיש להחזיר (חשוב לזכור שלפני הקריאה הראשונה לשיטה **MoveNext**, הערך המוחזר על ידי המאפיין **Current** אינו מוגדר, ויגרום לחריג). במקרה כזה, תהליך האתחול כולל יצירת מופע של התור וקריאה לשיטה **populate** שמטרתה לשלוף את הנתונים מהעץ ולהשתמש בהם למילוי התור.

לאחר הקריאה הראשונה, כל הקריאות לשיטה **MoveNext** גורמות למעבר בין פריט אחד למשנהו, עד אשר אין יותר פריטים. בדוגמה זו מסירים פריטים מהתור עד אשר התור ריק לגמרי. חשוב לזכור שהשיטה **MoveNext** לא מחזירה נתונים בפועל - המאפיין **Current** אחראי לביצוע פועלה זו. השיטה **MoveNext** מעדכנת את המצב הנוכחי ברשימה (מציבה במשתנה **currentItem** את הנתון שזה עתה נשלף מהתור) עבור המאפיין **Current**, ומחזירה **true** כאשר קיימים נתונים נוספים, או מחזירה **false** אם זה הנתון האחרון בתור.

11. שנה את הגדרת אחראי הגישה (accessor) get של המאפיין Current באופן הבא:

```
T IEnumerator<T>.Current
{
   get
   {
       if (this.enumData == null)
           throw new InvalidOperationException
           ("Use MoveNext before calling Current");
       return this.currentItem;
   }
}
```

המאפיין **Current** בודק את המשתנה **enumData** כדי לוודא שהתבצעה כבר קריאה לשיטה **MoveNext** (זכור שלפני הקריאה הראשונה ל-**MoveNext** ערכו של משתנה זה יהיה null). אם לא התבצעה קריאה, המאפיין זורק חריג **InvalidOperationException** - זהו מנגנון מקובל ביישומים של NET Framework שנועד לציין שלא ניתן לסיים פעולה מסוימת במצב הנוכחי. אם בוצעה כבר קריאה

לשיטה **MoveNext** אז השיטה כבר עדכנה את המשתנה **currentItem**, וכל מה שנותר למאפיין **Current** לעשות זה להחזיר את הערך שבמשתנה זה.

12. אתר את השיטה **IDisposable.Dispose**. הפוך את המשפט **throw new Exception** להערה. המונה אינו משתמש במשאבים המחייבים שחרור מפורש על ידי המתכנת. על כן, שיטה זו לא צריכה לעשות דבר, אך עליה להישאר בכל זאת בתוכנית. על השיטה **Dispose** למדנו בפרק 13.

13. בנה את הפתרון ותקן שגיאות במידת הצורך.

אתחול של משתנים שהוגדרו על ידי פרמטר-סוג

בוודאי הבחנת כבר שבמשפט המגדיר ומאתחל את המשתנה **currentItem** מופיעה מילת המפתח **default**. מילת מפתח זו היא מאפיין חדש אשר התווסף ל-C# 2.0.

המשתנה **currentItem** מוגדר על ידי פרמטר-סוג **T**. בעת הכתיבה וההידור של התוכנית ייתכן שהסוג הממשי אשר אמור להחליף את **T** איננו ידוע עדיין. עובדה זו מקשה על בחירת אופן אתחול המשתנה. הפתרון הקל יהיה לאתחל את המשתנה לערך **null**. עם זאת, אם הסוג המחליף את **T** הוא סוג ערך (value type), תהיה זו הצבה בלתי חוקית (לא ניתן להציב null בסוגי ערך, אלא רק בסוגי הפניה). כמו כן, אם תחליט להציב בו 0 מתוך הנחה שהערך יהיה נומרי, ההצבה לא תהיה חוקית אם הסוג יהיה סוג הפניה. יש אפשרויות פעולה נוספות. **T** יכול להיות ערך **בוליאני**, לדוגמה. מילת המפתח **default** פותרת את הבעיה הזו. הערך המשמש לאתחול המשתנה ייקבע בעת הפעלת המשפט. אם T יהיה מסוג הפניה, **default(T)** יחזיר null; אם T יהיה נומרי, **default(T)** יחזיר 0; אם T יהיה בוליאני, **default(T)** יחזיר **false**; אם **T** יהיה מסוג **מבנה**, כל אחד מהשדות השונים **במבנה** יאותחל באותו האופן (שדות הפניה יהיו **null**, שדות נומריים יהיו 0 ושדות בוליאניים יהיו **false**).

מימוש הממשק IEnumerable

בתרגיל הבא עליך לשנות את מחלקת העץ הבינארי כדי שתיישם את הממשק IEnumerable. השיטה **GetEnumerator** צריכה להחזיר אובייקט מסוג **TreeEnumerator<T>**.

ממש את הממשק IEnumerable<T> במחלקה Tree<T>

1. ב- Solution Explorer לחץ לחיצה כפולה על הקובץ Tree.cs כדי להציג את המחלקה **Tree<T>** בחלון Code and Text Editor.

2. שנה את הגדרת המחלקה **Tree<T>** כדי שתממש את הממשק **IEnumerable<T>** באופן הבא:

```
public class Tree<T> : IEnumerable<T> where T : IComparable<T>
```

שים לב שאת האילוצים יש להציב תמיד בסוף הגדרת המחלקה.

3. לחץ לחיצה ימנית על הממשק **IEnumerable<T>** שבהגדרת המחלקה, בתפריט הקיצור הצבע על Implement Interface ובחר Implement Interface Explicitly.

פעולה זו תיצור את השיטות **IEnumerable<T>.GetEnumerator** ושל **IEnumerable.GetEnumerator** ותוסיף אותם לקובץ. גם השיטה של הממשק IEnumerable נוצרת לאחר שהממשק **IEnumerable<T>** יורש מהממשק **IEnumerable**.

4. אתר את השיטה **IEnumerable<T>.GetEnumerator** שנמצאת בחלק האחרון של המחלקה. החלף את המשפט throw שבגוף השיטה **GetEnumerator()** עם הקוד הבא:

```
IEnumerator<T> IEnumerable<T>.GetEnumerator()
{
   return new TreeEnumerator<T>(this);
}
```

מטרת השיטה **GetEnumerator** לבנות מונה שצריך לדפדף באוסף. במקרה זה, צריך לבנות אובייקט חדש מסוג **TreeEnumerator<T>** באמצעות הנתונים שבעץ.

5. בנה את הפתרון.

במקרה זה אין לצפות לשגיאות בשלב ההידור. אם מופיעות שגיאות, תקן אותן ובנה מחדש את הפתרון.

כעת עליך להשתמש במשפט **foreach** להצגת תוכן העץ הבינארי, לבדיקת המחלקה **Tree<T>**.

בדוק את הרשימה

1. בתפריט File > Add בחר New Project. הוסף פרויקט חדש באמצעות התבנית Console Application. קרא לפרויקט **EnumeratorTest** ושמור אותו בתיקייה **Microsoft Press\Visual CSharp Step By Step\Chapter 18\BinaryTree** ולחץ OK.

2. לחץ לחיצה ימנית על הפרויקט EnumeratorTest שב- Solution Explorer ובתפריט הקיצור בחר Set as Startup Project.

3. בתפריט Project בחר Add Reference. בתיבת הדו-שיח Add Reference לחץ על הכרטיסייה Projects. בחר בפרויקט BinaryTree ולחץ OK.

האסופה (assembly) BinaryTree תופיע ברשימת ההפניות לפרויקט EnumeratorTest שנמצאת ב- Solution Explorer.

4. במחלקה Program שבחלון Code and Text Editor, הוסף לרשימה שבראש הקובץ את ההנחיה using הבאה:

```
using BinaryTree;
```

5. הוסף את המשפטים הבאים, שמטרתם ליצור ולמלא עץ בינארי של מספרים שלמים עבור השיטה **Main**:

```
Tree<int> tree1 = new Tree<int>(10);
tree1.Insert(5);
tree1.Insert(11);
tree1.Insert(5);
tree1.Insert(-12);
tree1.Insert(15);
tree1.Insert(0);
tree1.Insert(14);
tree1.Insert(-8);
tree1.Insert(10);
```

6. הוסף משפט **foreach** שתפקידו למנות את תכולת העץ ולהציג את התוצאות:

```
foreach (int data in tree1)
   Console.WriteLine(data);
```

7. בנה את הפתרון, ותקן טעויות במידת הצורך.

8. בתפריט Debug בחר Start Without Debugging.

התוכנית תופעל והערכים הבאים יופיעו על המסך בסדר זה (מימין לשמאל):

-12, -8, 0, 5, 5, 10, 10, 11, 14, 15

הקש Enter כדי לחזור אל Visual Studio 2005.

מימוש מונה באמצעות דפדפן

כפי שניתן לראות, הפיכת אוסף לאוסף שניתן לספירה (enumerable collection) עשויה להיות מורכבת ונתונה לשגיאות. כדי להקל על התהליך, C# 2.0 כולל איטרטורים אשר מבצעים באופן אוטומטי חלק גדול מהתהליך.

על פי המפרט של C# 2.0, **האיטרטור** (iterator) הוא בלוק של קוד אשר מניב **רצף** מסודר של ערכים. כמו כן, האיטרטור אינו בהכרח חבר במחלקה. הוא רק מגדיר את הרצף שלפיו אמור המונה להחזיר ערכים. במילים אחרות, האיטרטור מתאר את רצף הספירה כדי שהמהדר של C# יוכל ליצור מונה משלו. רעיון זה מורכב למדיי, ולכן נשתמש בדוגמה כדי להמחיש אותו לפני שנשוב לעצים הבינאריים ולנושא הרקורסיה.

איטרטור פשוט

המחלקה **BasicCollection<T>** שלהלן ממחישה את העקרונות הבסיסיים במימוש של איטרטור. המחלקה עושה שימוש ב-**List<T>** לאחסון נתונים, וכוללת את השיטה **FillList** כדי להזין נתונים לרשימה זו. שים לב שהמחלקה **BasicCollection<T>** מממשת את הממשק **IEnumerable<T>**. השיטה **GetEnumerator** ממומשת באמצעות האיטרטור:

```
class BasicCollection<T> : IEnumerable<T>
{
   private List<T> data = new List<T>();
   public void FillList(params T [] items)
   {
      for (int i = 0; i < items.Length; i++)
         data.Add(items[i]);
   }
   IEnumerator<T> IEnumerable<T>.GetEnumerator()
   {
      for (int i = 0; i < data.Count; i++)
         yield return data[i];
   }
   IEnumerator IEnumerable.GetEnumerator()
   {
      // Not implemented in this example
   }
}
```

השיטה **GetEnumerator** נראית לכאורה פשוטה למדיי, אולם יש לבחון אותה מקרוב. הדבר הראשון שעליך להתבונן בו הוא שהשיטה אינה מחזירה סוג **IEnumerator<T>**, אלא מבצעת לולאה אשר סוקרת את הפריטים השונים שבמערך **data**, ומחזירה כל אחד מהם בזה אחר זה. הנקודה העיקרית היא השימוש במילת המפתח **yield**, אשר מסמנת את הערך שיש להחזיר בכל איטרציה של הלולאה. ניתן לחשוב על המשפט **yield** כקריאה להפוגה זמנית של השיטה, והעברת הערך לגורם שקרא לשיטה. כאשר הגורם הקורא זקוק לערך הבא, השיטה **GetEnumerator ממשיכה לפעול בדיוק מאותה הנקודה שבה היא הפסיקה**, מבצעת לולאה נוספת ומניבה (yield) את הערך הבא. בסופו של דבר יועבר כל המידע, הלולאה תיפסק, והשיטה **GetEnumerator** תסתיים. כך מסתיימת למעשה האיטרציה בלולאה.

עליך לזכור שזו אינה שיטה רגילה. הקוד שבשיטה **GetEnumerator** מגדיר **iterator**. המהדר משתמש בקוד כדי לחולל מימוש של המחלקה **IEnumerator<T>** הכולל את השיטות **Current** ו-**MoveNext**. מימוש זה יבצע את הפעולות המפורטות בשיטה **GetEnumerator**. אינך יכול לראות את הקוד שמחולל המהדר, אלא אם כן תבצע decompile לאסופה (assembly, שהיא פעולה הפוכה של הידור הקוד לקובץ IL – כלומר השגת הקוד מתוך האסופה), אולם זהו מחיר מועט לשלם עבור השיפור בנוחות וצמצום הקוד שעליך לכתוב.

ניתן להפעיל את המונה אשר חולל האיטרטור באופן הבא:

```
BasicCollection<string> bc = new BasicCollection<string>();
bc.FillList("Twas", "brillig", "and", "the", slithy", "toves");
foreach (string word in bc)
   Console.WriteLine(word);
```

קוד זה יגרום להצגת תכולתו של האובייקט **bc** בסדר הבא (מימין לשמאל):

toves ,slithy ,the ,and ,brillig ,Twas

אם ברצונך ליצור מנגנוני דפדוף חילופיים אשר מציגים את המידע ברצף שונה, עליך לקבוע מאפיינים נוספים עבור הממשק **IEnumerable** אשר משתמשים בדפדפן כדי להחזיר נתונים. לדוגמה, המאפיין **Reverse** של המחלקה **BasicCollection<T>**, המוצג להלן, פולט את הנתונים בסדר הפוך:

```
public IEnumerable<T> Reverse
{
   get
   {
      for (int i = data.Count - 1; i >= 0; i--)
         yield return data[i];
   }
}
```

ניתן להפעיל מאפיין זה באופן הבא:

```
BasicCollection<string> bc = new BasicCollection<string>();
bc.FillList("Twas", "brillig", "and", "the", "slithy", "toves");
...
foreach (string word in bc.Reverse)
   Console.WriteLine(word);
```

קוד זה יגרום להצגת תכולת האובייקט **bc** בסדר הבא (מימין לשמאל):

toves, slithy, the, and, brillig, Twas.

הגדרת מונה עבור המחלקה Tree<T> באמצעות איטרטור

בתרגיל הבא עליך לממש מונה עבור המחלקה **Tree<T>** בעזרת איטרטור. בניגוד לתרגילים הקודמים שבהם העברנו את הנתונים שבעץ אל התור באמצעות השיטה **MoveNext**, כעת נוכל להגדיר דפדפן אשר יסרוק את העץ באמצעות מנגנון רקורסיבי טבעי יותר, המזכיר את השיטה **WalkTree** מפרק 17.

הוסף מונה למחלקה Tree<T>

1. הפעל את Microsoft Visual Studio 2005 אם טרם הפעלת.

2. פתח את הפתרון **\Microsoft Press\Visual CSharp Step By Step\Chapter 18\IteratorBinaryTree\BinaryTree.sln**. פתרון זה מכיל עותק עובד של הפרויקט BinaryTree אשר יצרת בפרק 17.

3. הצג את הקובץ Tree.cs בחלון Code and Text Editor. שנה את הגדרות המחלקה **Tree<T>** כדי לממש את הממשק **IEnumerable<T>**:

```
public class Tree<T> : IEnumerable<T> where T : IComparable<T>
{
    ...
}
```

4. לחץ לחיצה ימנית על הממשק **IEnumerable<T>** שבהגדרת המחלקה. בתפריט הקיצור הצבע על Implement Interface ובחר Implement Interface Explicitly.

   כתוצאה, השיטות **IEnumerable<T>.GetEnumerator** ו- **IEnumerable.GetEnumerator** יתווספו למחלקה.

5. אתר את השיטה **IEnumerable<T>.GetEnumerator**. שנה את תוכן השיטה **GetEnumerator** באופן הבא:

```
IEnumerator<T> IEnumerable<T>.GetEnumerator()
{
    if (this.LeftTree != null)
    {
        foreach (T item in this.LeftTree)
        {
            yield return item;
        }
    }
    yield return this.NodeData;
    if (this.RightTree != null)
    {
        foreach (T item in this.RightTree)
        {
            yield return item;
        }
    }
}
```

במבט ראשון זה לא נראה כך, אבל קוד זה הוא רקורסיבי. אם **LeftTree** אינו ריק, המשפט **foreach** הראשון מפעיל באופן מפורש את השיטה **GetEnumerator** (שאתה מגדיר כעת) על הענף. תהליך זה ממשיך עד אשר נמצא צומת ללא ענף שמאלי. בשלב זה, הקוד מניב את ערך המאפיין **NodeData**. בדיקה דומה מבוצעת על הענף הימני, וכאשר הוא נבדק בשלמותו, התהליך חוזר אל צומת השורש, פולט את המאפיין **NodeData** שלו, ובודק את הענף הימני שלו. תהליך זה ממשיך עד אשר כל העץ נבדק וכל הרכיבים הועברו כפלט.

בדוק את המונה החדש

1. בתפריט File, Add בחר Existing Project. פתח את התיקייה **Microsoft Press\Visual CSharp Step By Step\Chapter 18\BinaryTree\EnumeratorTest** ובחר את הקובץ EnumeratorTest.csproj. זהו הפרויקט אשר ייצרת קודם כדי לבדוק את המונה שפיתחת באופן ידני. לחץ Open.
2. לחץ לחיצה ימנית על הפרויקט EnumeratorTest שב- Solution Explorer ומתפריט הקיצור בחר Set as Startup Project.
3. ב- Solution Explorer, הרחב את התיקייה Reference של הפרויקט EnumeratorTest. לחץ לחיצה ימנית על האסופה BinaryTree, ומתפריט הקיצור בחר Remove.

   פעולה זו תגרום להסרת ההפניה לאסופה BinaryTree שבפרויקט.
4. בתפריט Project בחר Add Reference. בתיבת הדו-שיח Add Reference בחר בכרטיסייה Projects. סמן את הפרויקט BinaryTree ולחץ OK.

   האסופה BinaryTree תופיע כעת ברשימת ההפניות לפרויקט EnumeratorTest שב- Solution Explorer.

הערה

מטרת שני הצעדים האחרונים לוודא שהפרויקט EnumeratorTest מפנה אל הגרסה **החדשה** של האסופה BinaryTree, אשר משתמשת באיטרטור כדי ליצור את המונה, ולא לגרסה הקודמת.

5. עיין בשיטה **Main** שבקובץ Program.cs. זכור ששיטה זו יוצרת מופע של **Tree<T>**, מאחסנת בו מספר נתונים ומציגה את תכולתו באמצעות משפט **foreach**.
6. בנה את הפתרון ותקן שגיאות במידת הצורך.
7. בתפריט Debug בחר Start Without Debugging.

   כאשר תופעל התוכנית הנתונים אמורים להופיע בסדר זהה למקודם (מימין לשמאל): 12-, 8-, 0, 5, 5, 10, 10, 11, 14, 15
8. הקש Enter כדי לחזור אל Visual Studio 2005.

☯ **אם ברצונך להמשיך לפרק הבא**

השאר את Visual Studio 2005 פעילה, ועבור לפרק 19.

☯ **אם ברצונך לכבות כעת את Visual Studio 2005**

פתח את תפריט Menu ולחץ Exit. אם מופיעה תיבת דו-שיח Save, לחץ Yes.

פרק 18 – טבלה מסכמת

המשימה	צריך	דוגמה

להפוך מחלקה למחלקה שניתנת לספירה (enumerable class), וכך לאפשר לה לתמוך במבנה **foreach**.	לממש את הממשק **IEnumerable** וליצור שיטה **GetEnumerator** המחזירה אובייקט מסוג **IEnumerator**.	`public class Tree<T>:IEnumerable<T>`
		`{`
		`  ...`
		`  IEnumerator<T> GetEnumerator()`
		`  {`
		`  ...`
		`  }`
		`}`

לממש מונה (enumerator) ללא שימוש באיטרטור (iterator).	להגדיר מחלקת מונה לשם מימוש הממשק **IEnumerator**, ואשר כוללת את המאפיין **Current** ואת השיטות **MoveNext** ו-**Reset**.	`public class TreeEnumerator<T> : IEnumerator<T>`
		`{`
		`  ...`
		`  T Current`
		`  {`
		`  get`
		`  {`
		`  ...`
		`  }`
		`  }`
		`  bool MoveNext()`
		`  {`
		`  ...`
		`  }`
		`}`

להגדיר מונה באמצעות איטרטור.	לממש את המונה באמצעות משפט **yield** (תוצאה) שתפקידו לציין איזה פריטים יש להחזיר ובאיזה סדר.	`IEnumerator<T> GetEnumerator()`
		`{`
		`  for (...)`
		`  yield return ...`
		`}`

פרק 19

העמסת אופרטורים

לאחר סיום פרק זה, תוכל:

- ☯ לממש **אופרטורים בינאריים** עבור סוגים שהגדרת בעצמך.
- ☯ לממש **אופרטורים אונאריים** (unary) עבור סוגים שהגדרת בעצמך.
- ☯ לכתוב אופרטורים **הוספה** (increment) **והפחתה** (decrement) לסוגים שהגדרת בעצמך.
- ☯ להבין את הצורך במימוש של אופרטורים מסוימים כצמדים.
- ☯ לממש אופרטורים **המרה מרומזת** (implicit conversion operators) לסוגים שהגדרת בעצמך.
- ☯ לממש אופרטורים **המרה מפורשת** (explicit conversion operators) לסוגים שהגדרת בעצמך.

עד כה עשית שימוש רב בסימני האופרטורים הסטנדרטיים (כגון +, -) לביצוע פעולות סטנדרטיות (כגון חיבור וחיסור) על סוגים שונים של משתנים (כגון **int** ו-**double**). חלק גדול מהסוגים המובנים כוללים תגובות מוגדרות מראש לאופרטורים הללו. באפשרותך להגדיר אופרטורים עבור מחלקות ומבנים שיצרת בעצמך, וזהו הנושא של פרק זה.

מהם אופרטורים

מטרת האופרטורים לחבר בין שני אופרנדים כדי ליצור ביטוי. לכל אופרטור יש סמנטיקה ייחודית משלו המותנית בסוג המשתנים שהוא פועל עליהם או איתם. לדוגמה, משמעותו של האופרטור **+** היא "חיבור", כאשר הוא מופעל על סוגים נומריים, ומשמעותו "שרשור" כאשר הוא מופעל על מחרוזות. לכל אופרטור יש קדימות (precedence) משלו, אשר קובעת את סדר הביצוע שלו ביחס לאופרטורים אחרים. לדוגמה, הקדימות של האופרטור * גבוהה מזו של האופרטור +. משמעות הדבר היא, שתוצאת הביטוי a + b * c זהה לתוצאת הביטוי a+(b*c). כלומר, תחילה מבוצעת פעולת הכפל ואחר כך מבוצעת פעולת החיבור. לכל אופרטור יש גם **שיוך** (associativity) שנועד להגדיר אם הוא מחשב מימין לשמאל או משמאל לימין. לדוגמה, לאופרטור = יש שיוך ימני (כי הוא מחשב מימין לשמאל), ולכן a = b = c זהה לביטוי a = (b = c). כלומר, השיוך קובע את סדר ביצוע הפעולה, מימין לשמאל או משמאל לימין.

הערה

השיוך הימני של האופרטור = מאפשר לבצע מספר הצבות במשפט אחד. לדוגמה, ניתן לאתחל מספר משתנים לאותו הערך באופן הבא:

```
int a, b, c, d, e;
a = b = c = d = e = 99;
```

הביטוי הראשון שמחושב הוא e = 99. תוצאת הביטוי היא הערך המוצב (99), אשר לאחר מכן מוצב גם ב- d, c, b ולבסוף ב-a, בסדר הזה.

אופרטור **אונארי** (**unary**) פועל על אופרנד אחד בלבד. לדוגמה, אופרטור ההוספה (++) הוא אופרטור אונארי.

אופרטור **בינארי** (**binary**) הוא אופרטור הפועל על שני אופרנדים. לדוגמה, אופרטור הכפל (*) הוא אופרטור בינארי.

אילוצים של אופרטורים

שפת #C מאפשרת **להעמיס** (**overload**) את רוב האופרטורים הקיימים עם סוגים שהגדרת בעצמך. פעולה כזו תגרום לאופרטור שאתה כותב בעצמך להתווסף באופן אוטומטי למסגרת מוגדרת היטב, הכוללת את החוקים הבאים:

- אינך יכול לשנות את הקדימות והשיוך של אופרטור, אשר מבוססים על הסמל של האופרטור (כמו למשל +), ולא על הסוג (כמו למשל **int**) שעליו האופרטור מופעל. לפיכך, הביטוי a + b * c **לעולם** יהיה זהה לביטוי a + (b * c), ללא תלות בסוגי המשתנים המספריים **a**, **b** ו-**c**.
- אינך יכול להמציא סמלי אופרטורים חדשים. לדוגמה, אינך יכול לקבוע שהסימן ** ישמש להעלאת מספר אחד בחזקת מספר אחר. לביצוע פעולה זו עליך להפעיל שיטה (method).
- אינך יכול לשנות את משמעות האופרטורים כאשר הם מיושמים על סוגים מובנים. לדוגמה, לביטוי 2 + 1 יש משמעות מוגדרת מראש ואינך יכול לעקוף או לשנות את המשמעות הזו.
- יש אופרטורים שלא ניתן להעמיס. לדוגמה, אינך יכול להעמיס את אופרטור הנקודה (גישה לחברים). גם ההגבלה הזו הינה לטובה, מכיוון שהיכולת להעמיס את האופרטורים הללו הייתה יוצרת סיבוכים מיותרים.

טיפ

ניתן להשתמש בסדרנים (indexers) כדי לדמות את הסימון **[ ]** כאופרטור; ניתן להשתמש במאפיינים (properties) כדי לדמות את = כאופרטור; וניתן להשתמש בנציגים (delegates) כדי לדמות קריאה לפונקציה כאופרטור.

אופרטורים מועמסים

כדי להגדיר התנהגות של אופרטורים בעצמך, עליך "להעמיס אופרטור". כדי לעשות זאת יש להשתמש בתחביר הדומה לזה שמשמש להעמסת שיטות, עם ערך מוחזר ופרמטרים, אולם במקום שם השיטה יש לרשום את מילת המפתח **operator** לצד סמל האופרטור שעליו אתה מכריז. לדוגמה, לפניך מבנה (struct) בשם **Hour** שהוגדר על ידי המשתמש ואשר מגדיר אופרטור + בינארי לשם חיבור של שני מופעים של **Hour**.

```
struct Hour
{
   public Hour(int initialValue)
   {
       this.value = initialValue;
   }
   public static Hour operator+ (Hour lhs, Hour rhs)
   {
       return new Hour(lhs.value + rhs.value);
   }
   ...
   private int value;
}
```

להלן השימוש באופרטור:

```
Hour firstHour = new Hour(2);
Hour secondHour = new Hour(3);
Hour thirdHour = firstHour + secondHour; // thirdHour = 5
```

שים לב לנקודות הבאות:

- האופרטור הוא **ציבורי** (public). כל האופרטורים **חייבים** להיות ציבוריים.
- האופרטור הוא **סטטי** (static). כל האופרטורים **חייבים** להיות סטטיים, ולא ניתן להחיל עליהם את מילות המפתח **virtual**, **abstract**, **override** או **sealed**.
- לאופרטור בינארי (כגון + המוצג לעיל) יש שני ארגומנטים מפורשים, בעוד שלאופרטור אונארי יש ארגומנט מפורש אחד (מתכנתים מרקע של C++ צריכים לשים לב שלאופרטורים אין לעולם פרמטר **this** מוסתר).

טיפ

כאשר מכריזים על פעולות מסוגננות ביותר (כגון אופרטורים), רצוי לנהוג לפי המוסכמות בקביעת שמות לפרמטרים. לדוגמה, הפרמטרים של אופרטורים בינאריים נקראים לעיתים קרובות **rhs** ו-**lhs** (ראשי תיבות של right-hand side ו-left-hand side בהתאמה).

כאשר תשתמש באופרטור + על שני ביטויים מסוג **Hour**, המהדר של C# ימיר באופן אוטומטי את האופרטור שלך בקריאה לקוד של האופרטור. הנה דוגמה לפקודות המקוריות ולקוד המעודכן:

```
Hour Example(Hour a, Hour b)
{
   return a + b;
}
```

הקוד המעודכן:

```
Hour Example(Hour a, Hour b)
{
   return Hour.operator+(a,b); // pseudocode
}
```

שים לב שתחביר זה הוא פסבדו-קוד, ולא קוד תקני של C#. בעת הפעלה של אופרטור בינארי ניתן להשתמש רק בסימון הסטנדרטי, שבו סמל האופרטור נמצא בין שני אופרנדים.

יש כלל אחד נוסף שעליך לקיימו בעת הכרזה על אופרטור, ואם לא תנהג לפיו, הידור הקוד ייכשל: לפחות אחד מהפרמטרים חייב להיות מהסוג שהוגדר. בדוגמה הקודמת של **operator+** עבור המחלקה **Hour**, אחד מהפרמטרים, **a** או **b**, צריך להיות אובייקט מסוג **Hour**. בדוגמה זו שני הפרמטרים היו מסוג **Hour**. עם זאת, ייתכנו מצבים שבהם תרצה להגדיר גרסאות נוספות של **operator+** אשר מוסיפות מספר שלם (כמו מספר שעות) לאובייקט מסוג **Hour**. במקרה זה, הפרמטר הראשון יהיה מסוג **Hour** והפרמטר השני יכול להיות מספר שלם. כלל זה מקל על המהדר לקרוא לגרסה הנכונה של האופרטור, וגם מבטיח שלא יהיה ניתן לשנות את משמעות האופרטורים המובנים.

יצירת אופרטורים סימטריים

בסעיף קודם למדת כיצד להכריז על אופרטור **+** בינארי, כדי לחבר יחד שני מופעים של הסוג **Hour**. המבנה **Hour** גם מכיל בנאי היוצר **Hour** מ-**int**. משמעות הדבר היא שניתן גם לחבר **Hour** ו-**int**, לשם כך, עליך להשתמש בבנאי של **Hour** כדי להמיר את **int** ל-**Hour** לפני הפעלת האופרטור. הנה דוגמה:

```
Hour a = ...;
int b = ...;
Hour sum = a + new Hour(b);
```

אין ספק שזהו קוד תקני, אולם הוא אינו ברור ותמציתי כמו חיבור ישיר של Hour ו-int באופן הבא:

```
Hour a = ...;
int b = ...;
Hour sum = a + b;
```

כדי שהביטוי (a + b) יהיה תקני, עליך להגדיר את המשמעות של חיבור **Hour** (**a**, מצד שמאל) עם **int** (**b**, מצד ימין). במילים אחרות, עליך להכריז על אופרטור + בינארי אשר הפרמטר הראשון שלו הוא מסוג **Hour** והפרמטר השני הוא מסוג **int**. הקוד שלהלן מדגים את הדרך המומלצת לעשות זאת:

```
struct Hour
{
   public Hour(int initialValue)
   {
       this.value = initialValue;
   }
   ...
   public static Hour operator+ (Hour lhs, Hour rhs)
   {
       return new Hour(lhs.value + rhs.value);
   }
   public static Hour operator+ (Hour lhs, int rhs)
   {
       return lhs + new Hour(rhs);
   }
```

```
    ...
    private int value;
}
```

שים לב שהגרסה השנייה של אופרטור זה יוצרת אובייקט **Hour** מהארגומנט **int**, ואז קוראת לגרסה הראשונה. בדרך זו נשמר העיקרון של השימוש באופרטור. אנו רואים ש-**operator+** נוסף מקל על ביצוע הפעולות הקיימות. כמו כן, שים לב שאין צורך ליצור גרסאות שונות של האופרטור שכל אחת מהן מופעלת עבור סוג אחר של פרמטר שני. צריך ליצור גרסאות של הפרמטר רק עבור המקרים העיקריים, ולתת למשתמש במחלקה לנקוט בצעדים הנחוצים במקרים יוצאי דופן.

operator+ מגדיר כיצד יש לחבר יחדיו אופרטור שמאלי **Hour** ואופרטור ימני **int**, אולם הוא אינו קובע כיצד יש לחבר יחד אופרטור שמאלי **int** ואופרטור ימני **Hour**:

```
int a = ...;
Hour b = ...;
Hour sum = a + b; // compile-time error
```

פעולה זו מהווה תגובה אינטואיטיבית. אם ניתן לכתוב את הביטוי a + b, מן הסתם גם ניתן לכתוב b + a. אבל, לשם כך עליך ליצור העמסה נוספת של **operator+**:

```
struct Hour
{
    public Hour(int initialValue)
    {
        this.value = initialValue;
    }
    ...
    public static Hour operator+ (Hour lhs, int rhs)
    {
        return lhs + new Hour(rhs);
    }
    public static Hour operator+ (int lhs, Hour rhs)
    {
        return new Hour(lhs) + rhs;
    }
    ...
    private int value;
}
```

הערה

מתכנתים מרקע של C++ צריכים לשים לב שיש ליצור את ההעמסה לבד. המהדר לא יצור אותה עבורך, וגם לא יחליף אוטומטית בין שני האופרנדים כדי לנסות למצוא אופרטור מתאים.

אופרטורים ומפרט השפה המשותפת (CLS)

לא כל שפות התכנות מנצלות את תמיכת השפה המשותפת CLS (Common Language Specification) של סביבת ההרצה CLR (Common Language Runtime), ולא כולן יכולות להבין העמסה של אופרטורים. לכן, אחת מהדרישות של מפרט השפה המשותפת CLS היא, שבעת העמסה של אופרטור עליך ליצור מנגנון חילופי לפעולה הנתמכת על ידי ה-CLR. לדוגמה, נניח שאתה מפעיל את **operator+** עבור המבנה **struct**:

```
public static Hour operator+ (Hour lhs, int rhs)
{
   ...
}
```

עליך ליצור גם שיטה **Add** המבצעת את אותה הפעולה:

```
public static Hour Add(Hour lhs, int rhs)
{
   ...
}
```

הבנת הצבה מורכבת

שפת #C אינה מאפשרת להכריז על אופרטורים של הצבה אשר הגדרת בעצמך. עם זאת, אופרטור של הצבה מורכבת (כגון =+) מחושב תמיד באמצעות האופרטור המיוחס אליו (כמו +). במילים אחרות, הביטוי הזה:

```
a += b;
```

מחושב אוטומטית באופן הבא:

```
a = a + b;
```

למעשה, הביטוי a @= b (שבו @ מייצג אופרטור תקני) מחושב תמיד כך: a = a @ b. כל עוד הכרזת על אופרטור פשוט תקני, הוא תמיד ינחה את החישוב כאשר נעשה שימוש באופרטור הצבה מורכבת המיוחס לו. לדוגמה:

```
Hour a = ...;
int b = ...;
a += a; // same as a = a + a
a += b; // same as a = a + b
```

ביטוי ההצבה המורכבת הראשון (a += a) הינו תקני מכיוון ש-a הוא מסוג **Hour**, והסוג **Hour** מכיל **operator+** בינארי אשר שני הפרמטרים שלו הם מסוג **Hour**. גם ביטוי ההצבה המורכבת השני (a += b) הוא תקני, מכיוון ש-**a** הוא מסוג **Hour** ו-**b** הוא מסוג **int**. הסוג **Hour** גם מכיל **operator+** בינארי, אשר הפרמטר הראשון שלו הוא מסוג **Hour** והפרמטר השני הוא מסוג **int**. אולם לא ניתן לכתוב את הביטוי b += a, מכיוון שהביטוי המקביל אליו הוא b = b + a. למרות שהחיבור תקני, ההצבה אינה תקנית כי ניתן להציב **Hour** בסוג **int** המובנה.

הכרזה על אופרטורים מוסיפים ומפחיתים

שפת התכנות #C מאפשרת לך להגדיר גרסאות לאופרטורי ההוספה (++) וההפחתה (--). הכללים הרגילים תקפים גם לגבי העמסת האופרטורים הללו: עליהם להיות ציבוריים, עליהם להיות סטטיים ועליהם להיות אונאריים. זהו אופרטור ההוספה עבור המבנה **Hour**:

```
struct Hour
{
   ...
   public static Hour operator++ (Hour arg)
   {
      arg.value++;
      return arg;
   }
   ...
   private int value;
}
```

האופרטורים של ההוספה ושל ההפחתה הם ייחודיים, במובן זה שניתן להשתמש בהם לפני ואחרי האופרנד. שפת #C עושה שימוש נבון באותו האופרטור בשתי הגרסאות השונות. ביטוי שבו האופרטור נמצא אחרי האופרנד (postfix) יחזיר את ערך האופרנד **לפני** הפעלת הביטוי. או במילים אחרות, המהדר ממיר את הביטוי הזה:

```
Hour now = new Hour(9);
Hour postfix = now++;
```

ומקבלים את הביטוי הבא:

```
Hour now = new Hour(9);
Hour postfix = now;
now = Hour.operator++(now); // pseudocode, not valid C#
```

ביטוי שבו האופרטור נמצא לפני האופרנד (prefix) יחזיר את ערך האופרנד אחרי הפעלת הביטוי. המהדר ממיר את הביטוי הזה:

```
Hour now = new Hour(9);
Hour prefix = ++now;
```

ומקבלים את הביטוי הבא:

```
Hour now = new Hour(9);
now = Hour.operator++(now); // pseudocode, not valid C#
Hour prefix = now;
```

משמעות הדבר היא שסוג הערך המוחזר של אופרטור ההוספה ואופרטור ההפחתה חייב להיות זהה לסוג הפרמטר.

אופרטורים במבנים ובמחלקות

חשוב להבין שהמימוש של אופרטור ההוספה במבנה **Hour** פועל כהלכה רק מכיוון ש-**Hour** הוא מבנה (struct). אם תהפוך את **Hour** למחלקה ולא תשנה את אופן המימוש של אופרטור ההוספה, גרסת postfix לא תחזיר את הערך הנכון. כדי להבין את המקור לטעות צריך לזכור שמחלקה היא מסוג הפניה. לשם כך רצוי לעיין שוב בפריסת הביטוי שהמהדר מחולל:

```
Hour now = new Hour(9);
Hour postfix = now;
now = Hour.operator++(now); // pseudocode, not valid C#
```

אם **Hour** הוא מסוג מחלקה, משפט ההצבה postfix = now גורם למשתנה **postfix** להפנות אל האובייקט שאליו מפנה **now**. לכן, כל שינוי של **now** יבוא מייד לידי ביטוי ב-**prefix**. אם **Hour** הוא מסוג מבנה, משפט ההצבה יוצר עותק של **now** ב-**prefix**, והשינויים שמתבצעים ב-**now** אינם משפיעים כלל על **prefix**.

המימוש הנכון של אופרטור ההוספה, כאשר Hour הוא מסוג מחלקה, הוא:

```
class Hour
{
   public Hour(int initialValue)
   {
      this.value = initialValue;
   }
   ...
   public static Hour operator++(Hour arg)
   {
      return new Hour(arg.value + 1);
   }
   ...
   private int value;
}
```

שים לב שכעת **operator++** יוצר אובייקט חדש שמתבסס על הנתונים שבאובייקט המקורי. הנתונים שבאובייקט החדש גדלים באחד, בעוד שהנתונים שבאובייקט המקורי אינם משתנים. שיטה זו פועלת כראוי, אך בדרך זו נוצר אובייקט חדש בכל הפעלה של אופרטור ההוספה. הדבר עלול להיות בזבזני מבחינת ניצול הזיכרון והמשאבים שמוקצים לפינוי אשפה. לכן, רצוי להימנע ככל שאפשר מהעמסת אופרטורים בעת הגדרת מחלקות.

הגדרת צמדים של אופרטורים

אופרטורים מסוימים באים באופן טבעי בזוגות. לדוגמה, אם ניתן להשוות בין שני ערכי **Hour** באמצעות האופרטור **!=**, מן הסתם ניתן גם להשוות בין שני ערכי **Hour** באמצעות האופרטור ==. המהדר של C# מאלץ אותך לעמוד בציפיות אלו. כאשר אתה מגדיר את אחד האופרטורים **!=** או ==, אתה חייב להגדיר גם את האחר. החוק של "הכל או כלום", תקף גם לאופרטורים

< ו- > ולאופרטורים =< ו- =>. המהדר לא יגדיר באופן אוטומטי עבורך את האופרטור שלא הגדרת בכוונה או בטעות. עליך להקפיד לעשות זאת באופן מפורש בעצמך, ולא לסמוך על כך שהדבר ברור מאליו. לפניך הגדרת האופרטורים =! ו- == עבור המבנה **Hour**:

```
struct Hour
{
   public Hour(int initialValue)
   {
      this.value = initialValue;
   }
   ...
   public static bool operator==(Hour lhs, Hour rhs)
   {
      return lhs.value == rhs.value;
   }
   public static bool operator!=(Hour lhs, Hour rhs)
   {
      return lhs.value != rhs.value;
   }
   ...
   private int value;
}
```

סוג הערך המוחזר של האופרטורים הללו אינו חייב בהכרח להיות בוליאני. אולם, כדאי שתהיה לך סיבה ממש טובה כדי להגדיר סוג ערך מוחזר אחר עבור האופרטורים הללו, אחרת הדבר עשוי להיות מאוד מבלבל.

הערה

אם אתה מגדיר **אופרטור** == ו**אופרטור** =!, עליך לעקוף (override) את השיטות **Equals** ו-**GetHashCode** העוברות בירושה מ- **System.Object**. השיטה **Equals** נועדה לבצע פעולה זהה לזו של ה**אופרטור** == (להגדיר פריט אחד ביחס לאחר). השיטה **GetHashCode** משמשת שיטות אחרות ב-NET Framework.. לדוגמה, כאשר אובייקט כלשהו משמש בתור מפתח ב-hash table, השיטה **GetHashCode** מופעלת על האובייקט, כדי לעזור בחישוב הערך בתוך הטבלה הזו. למידע נוסף בנושא עיין ב-NET Framework Reference documentation שכלול ב- Visual Studio 2005.

מימוש אופרטור

בתרגיל הבא עליך לכתוב יישום של שעון דיגיטלי נוסף הפועל בסביבת Microsoft Windows. גרסה זו של הקוד דומה לתרגיל שבפרק 16, אך כעת השיטה **delegate** המופעלת בכל שנייה, אינה מקבלת את הערכים הנוכחיים של **hour**, **minute** ו-**second**. כאן היא מעדכנת את השעה הנוכחית באמצעות עדכון של שלושה שדות, עבור כל אחד מהערכים **hour**, **minute** ו-**second**. שלושת השדות הללו הם מסוג **hour**, **minute** ו-**second** בהתאמה, וכולם מבנים. למרות זאת, היישום עדיין לא יכול לעבור הידור, מכיוון שהמבנה **Minute** אינו גמור. בתרגיל הראשון, עליך לכתוב את האופרטורים החסרים במבנה **Minute** כדי לסיים את קידוד המבנה.

כתוב את ההעמסות של operator+

1. הפעל את Microsoft Visual Studio 2005 אם אינה פועלת עדיין.
2. פתח את הפרויקט **Operators** שבתיקייה My Documents\Microsoft Press\Visual CSharp Step by Step\Chapter 19\Operators.
3. הצג את קובץ המקור Clock.cs בחלון Code and Text Editor, ואתר את ההגדרות של השדות **hour**, **minute** ו-**second** בסוף המחלקה.

   שדות אלו מאחסנים את השעה הנוכחית:

```
class Clock
{
   ...
   private Hour hour;
   private Minute minute;
   private Second second;
}
```

4. אתר את השיטה **tock** של המחלקה Clock. שיטה זו מופעלת בכל שנייה כדי לעדכן את המשתנים hour, minute ו-second.

```
private void tock()
{
   this.second++;
   if (this.second == 0)
   {
      this.minute++;
      if (this.minute == 0)
      {
         this.hour++;
      }
   }
}
```

הבנאים של המחלקה **Clock** כוללים את המשפטים הבאים שתפקידם לדאוג לכך שתבוצע קריאה לשיטה זו בכל פעם שמוצף האירוע **tick** של השדה **pulsed**. שים לב שהשדה **pulsed** הוא מסוג **Ticker**, ומשתמש באובייקט מסוג **Timer** כדי לחולל אירוע בכל שנייה, כמוסבר בפרק 16.

```
this.pulsed.tick += tock;
```

5. בתפריט Build בחר Build Solution.

   סיימת את הפרויקט, אך הוא יציג את הודעת השגיאה הבאה בחלונית Output:

```
Operator '==' cannot be applied to operands of type 'Operators.
Minute' and 'int'.
```

הבעיה היא, שבמבנה **Minute** חסרה הכרזה מתאימה על ==**operator**, השיטה **tock** מכילה את משפט **if** הבא:

```
if (minute == 0)
{
   hour++;
}
```

משימתך הבאה היא לקודד את האופרטור עבור המבנה **Minute**.

6. בחלון Code and Text Editor, פתח את הקובץ Minute.cs.

7. בחלון Code and Text Editor, כתוב גרסה של ==**operator** אשר מקבלת אופרנד שמאלי מסוג **Minute** ואופרנד ימני מסוג **int**. אל תשכח שהערך המוחזר של אופרטור זה צריך להיות מסוג bool.

האופרטור המושלם צריך להיראות בדיוק כמו במחלקה הבאה:

```
struct Minute
{
   ...
   public static bool operator== (Minute lhs, int rhs)
   {
      return lhs.value == rhs;
   }
   ...
   private int value;
}
```

8. בתפריט Build בחר Build Solution.

הפרויקט מוכן אבל יציג את הודעת השגיאה הבאה בחלונית Output:

```
The operator 'Operators.Minute.operator ==(Operators.Minute, int)'
 requires a matching operator "!=" to also be defined.
```

היישום אינו פועל עדיין. הבעיה היא שלמרות שכתבת גרסה מתאימה ל-==**operator**, לא כתבת את הגרסה עבור בן הזוג =**operator!**.

9. כתוב גרסה של =operator! אשר מקבלת אופרנד שמאלי מסוג Minute ואופרנד ימני מסוג int.

האופרטור המושלם נראה בדיוק כך:

```
struct Minute
{
   ...
   public static bool operator!= (Minute lhs, int rhs)
   {
      return lhs.value != rhs;
   }
   ...
   private int value;
}
```

10. בתפריט Build בחר Build Solution.
הפעם היישום מוכן ללא שגיאות.
11. בתפריט Debug בחר Start Without Debugging.
היישום פועל ומציג שעון דיגיטלי המעדכן את עצמו בכל שנייה.
12. סגור את היישום וחזור לסביבת התכנות של Visual Studio 2005.

אופרטורים של המרה

לעיתים צריך להמיר ביטוי מסוג מסוים לביטוי מסוג אחר. לדוגמה, לפי ההכרזה של השיטה הבאה, היא מקבלת פרמטר אחד מסוג **double**:

```
class Example
{
   public static void MyDoubleMethod(double parameter)
   {
   ...
   }
}
```

כביכול, בעת קריאה לשיטה **MyDoubleMethod** ניתן להעביר לה ארגומנטים מסוג **double** בלבד, אולם אין זה נכון. המהדר של #C מאפשר לקרוא לשיטה **MyDoubleMethod** גם עם ארגומנטים שאינם מסוג **double**, כל עוד ניתן להמיר את ערכם ל-**double**. המהדר יחולל קוד אשר מבצע את ההמרה בעת הקריאה לשיטה.

יצירת המרות מובנות

הסוגים המובנים כוללים המרות מובנות. לדוגמה, ניתן להמיר באופן מרומז סוג **int** ל-**double**. ביצוע **המרה מרומזת** (implicit conversion) אינו דורש תחביר מיוחד ולעולם לא יגרום לזריקת חריג:

```
Example.MyDoubleMethod(42); // implicit int to double conversion
```

המרה מרומזת נקראת לעיתים **המרה מרחיבה** (**widening conversion**), מכיוון שתוצאת ההמרה תמיד רחבה יותר מהערך המקורי. היא מכילה כמות מידע גדולה או שווה לזו של הערך המקורי, ולא אובד דבר.

מצד שני, לא ניתן להמיר באופן מרומז סוג **double** ל-**int**:

```
class Example
{
   public static void MyIntMethod(int parameter)
   {
      ...
   }
}
```

```
...
Example.MyIntMethod(42.0); // compile-time error
```

יש סכנה של אובדן מידע בהמרה של **double** ל-**int**, ולכן ההמרה לא תבוצע אוטומטית. תוכל לבדוק בעצמך מה קורה אם הארגומנט של השיטה **MyDoubleMethod** היה 42.5 - כיצד הייתה מבוצעת ההמרה? עם זאת, ניתן להמיר סוג **double** ל-**int**, אולם כדי לעשות זאת צריך להשתמש בסימון מפורש (cast):

```
Example.MyIntMethod((int)42.0);
```

המרה מפורשת (**explicit conversion**) נקראת לעיתים **narrowing conversion**, מכיוון שהתוצאה פחותה יותר מהערך המקורי (היא יכולה להכיל פחות נתונים), ועשויה לזרוק את החריג **OverflowException**. שפת #C מאפשרת ליצור אופרטורים של המרה עבור סוגי נתונים שהגדרת בעצמך כדי שתוכל לשלוט ביכולת להמירם באופן מרומז או מפורש לסוגים אחרים.

מימוש אופרטורים של המרה המוגדרים על ידי המשתמש

התחביר המשמש להכרזה על אופרטורים של המרה המוגדרים על ידי המשתמש, דומה לתחביר המשמש להעמסת אופרטורים. אופרטור המרה חייב להיות ציבורי וגם חייב להיות סטטי. להלן אופרטור המרה המאפשר לבצע המרה מרומזת (implicit) של אובייקט מסוג Hour ל-**int**:

```
struct Hour
{
   ...
   public static implicit operator int (Hour from)
   {
      return this.value;
   }
   private int value;
}
```

הסוג שאתה ממיר משמש כפרמטר היחידי (במקרה זה **Hour**), והסוג שאליו אתה ממיר משמש כשם הסוג לאחר מילת המפתח **operator** (במקרה זה **int**). אין צורך לציין את סוג הערך המוחזר לפני מילת המפתח **operator**.

בעת הכרזה על אופרטורים של המרה, עליך לציין אם הם אופרטורים של המרה מרומזת או מפורשת. תוכל לעשות זאת באמצעות מילות המפתח **implicit** ו-**explicit** בהתאמה. לדוגמה, אופרטור ההמרה של **Hour** ל-**int** שזה עתה הוצג הוא implicit, ולכן המהדר של #C יכול להשתמש בו באופן מרומז (ללא השלכה - cast):

```
class Example
{
   public static void Method(int parameter) { ... }
   public static void Main()
   {
      Hour lunch = new Hour(12);
      Example.MyOtherMethod(lunch); // implicit Hour to int conversion
   }
}
```

אם אופרטור ההמרה הוגדר כ-**explicit**, לא היה ניתן להדר את הדוגמה האחרונה, מכיוון שאופרטור המרה מפורש מחייב השלכה מפורשת:

```
Example.MyOtherMethod((int)lunch); // explicit Hour to int conversion
```

מהם השיקולים לבחירה בין המרה מפורשת לבין המרה מרומזת? אם ההמרה תמיד בטוחה, ואין בה סיכון של אובדן מידע או של זריקת חריג, ניתן להגדיר את ההמרה כמרומזת. אחרת יש להכריז עליה כהמרה **מפורשת**. המרת סוג **Hour** ל-**int** תמיד בטוחה - לכל שעה יש ערך שלם המקביל אליה - ולכן אין קושי בהכרזה עליה כמרומזת. אולם במקרה של אופרטור שתפקידו להמיר **string** ל-**Hour**, כדאי להכריז על ההמרה כמפורשת, מכיוון שלא כל המחרוזות מייצגות שעה תקנית. שים לב שבעוד שמחרוזת "7" היא תקנית, את המחרוזת "Hello, World", למשל, לא ניתן להמיר ל**שעה**.

יצירת אופרטורים סימטריים משופרים

אופרטורים של המרה מאפשרים לפתור את הקושי שביצירת אופרטורים סימטריים בצורה פשוטה יותר. לדוגמה, במקום ליצור שלוש גרסאות של **operator+** (**Hour** + **Hour**, **Hour** + **int**, **int** + **Hour**) עבור המבנה **Hour** כפי שהוצג קודם לכן, ניתן ליצור גרסה אחת של **operator+** (המקבלת שני פרמטרים מסוג **Hour**) והמרה מרומזת של **int** ל-**Hour**, באופן הבא:

```
struct Hour
{
   public Hour(int initialValue)
   {
      this.value = initialValue;
   }
   public static Hour operator+(Hour lhs, Hour rhs)
   {
      return new Hour(lhs.value + rhs.value);
   }
   public static implicit operator Hour (int from)
   {
      return new Hour (from);
   }
   ...
   private int value;
}
```

אם אתה מחבר Hour ו-int (בכל סדר שהוא), המהדר של C# ממיר באופן אוטומטי את int ל-Hour ולאחר מכן קורא לאופרטור +operator עם שני ארגומנטים מסוג Hour:

```
void Example(Hour a, int b)
{
   Hour eg1 = a + b; // b converted to an Hour
   Hour eg2 = b + a; // b converted to an Hour
}
```

הוספת אופרטור של המרה מרומזת

בתרגיל הבא עליך לשנות את יישום השעון הדיגיטלי שהופיע בתרגיל האחרון. עליך להוסיף אופרטור של המרה מרומזת אל המבנה **Second** ולהסיר את האופרטורים שהוא מחליף.

כתוב אופרטור המרה

1. חזור אל Visual Studio 2005 שבה תמצא את הפרויקט Operators. הצג את הקובץ Clock.cs בחלון Code and Text Editor ועיין שוב בשיטה tock:

```
private void tock()
{
   this.second++;
   if (this.second == 0)
   {
      this.minute++;
      if (this.minute == 0)
      {
         this.hour++;
      }
   }
}
```

שים לב למשפט if (this.second == 0). חלק זה של הקוד משווה בין **Second** לבין **int** באמצעות האופרטור ==.

2. בחלונית Code, פתח את קובץ המקור Second.cs.

3. בחלון Code and Text Editor, מחק את הגרסאות של **operator**== ושל **operator!**= המקבלות פרמטר אחד מסוג **Second** ופרמטר נוסף מסוג **int**. אל תמחק בטעות את האופרטורים המקבלים שני פרמטרים מסוג **Second**. שלוש הגרסאות הבאות הן היחידות הדרושות למימוש של operator== ושל operator!= במבנה **Second**:

```
struct Second
{
   ...
   public static bool operator==(Second lhs, Second rhs)
   {
      return lhs.value == rhs.value;
   }
   public static bool operator!=(Second lhs, Second rhs)
   {
      return lhs.value != rhs.value;
   }
   public static bool operator++(Second lhs, Second rhs)
   {
      return lhs.value ++ rhs.value;
   }
      ...
}
```

4. בתפריט Build בחר Build Solution.

הבנייה תיכשל, וההודעה הבאה תופיע על המסך:

```
Operator ‘==’ cannot be applied to the operands of type
 ‘Operators.Second’ and ‘int’
```

המשמעות היא שהסרת האופרטורים המשווים בין **Second** לבין int גרמה לשגיאת הידור.

5. בחלון Code and Text Editor, הוסף אופרטור של המרה מרומזת למבנה **Second** שתפקידו להמיר סוג **int** ל-**Second**.

אופרטור ההמרה נראה כך:

```
struct Second
{
   ...
   public static implicit operator Second (int arg)
   {
      return new Second(arg);
   }
   ...
}
```

6. בתפריט Build בחר Build Solution ותקן שגיאות במידת הצורך.

התוכנית תצליח להיבנות הפעם, מכיוון שאופרטור ההמרה בשילוב עם שני האופרטורים הנותרים יכולים לבצע את הפעולות שמבוצעות על ידי ארבע ההעמסות שמחקנו בתחילת התרגיל. ההבדל היחיד הוא ששימוש באופרטור המרה מרומזת עשוי להיות איטי במקצת לעומת החלופה האחרת.

7. בתפריט Debug בחר Start Without Debugging.

ודא שהיישום עדיין פועל.

8. סגור את היישום וחזור לסביבת התכנות של Visual Studio 2005.

☯ **אם ברצונך להמשיך לפרק הבא**

השאר את Visual Studio 2005 פועלת ועבור לפרק 20.

☯ **אם ברצונך לכבות כעת את Visual Studio 2005**

פתח את תפריט Menu ולחץ Exit. אם מופיעה תיבת דו-שיח Save, לחץ Yes.

פרק 19 – טבלה מסכמת

המשימה	צריך

לכתוב אופרטור.	לכתוב את מילות המפתח public ו-static, אחריהן את סוג הערך המוחזר, אחריו את מילת המפתח operator, אחריו את הסמל של האופרטור המוכרז ולבסוף הפרמטרים המתאימים בין צמד סוגריים. הנה דוגמה:
	`struct Hour`
	`{`
	`...`
	`public static bool operator==(Hour lhs, Hour rhs)s`
	`{`
	`...`
	`}`
	`...`
	`}`

להכריז על אופרטור המרה.	לכתוב את מילות המפתח public ו-static, אחריהן את מילת המפתח implicit או explicit, אחריהן את מילת המפתח operator, אחר כך את סוג הערך שאליו ממירים, ולבסוף לכתוב את סוג הערך שממנו ממירים כפרמטר בודד בין צמד סוגריים. לדוגמה:
	`struct Hour`
	`{`
	`...`
	`public static implicit operator Hour(int arg)`
	`{`
	`...`
	`}`
	`...`
	`}`

חלק ד'

עבודה עם יישומי Windows

בחלק זה:

פרק 20

מבוא – Windows Forms

לאחר סיום פרק זה, תוכל:

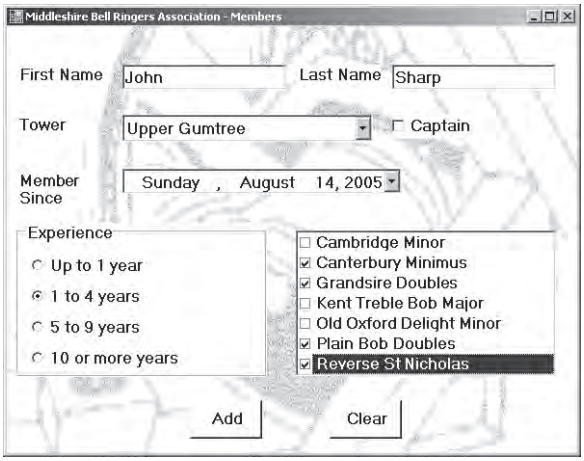
- ליצור יישום המבוסס על **טפסי Windows** (Windows Forms).
- להשתמש בפקדי Windows Forms משותפים, כגון **תוויות** (labels), **תיבות טקסט** (textboxes) **ולחצנים** (buttons).
- לשנות את המאפיינים של Windows Forms והפקדים בזמן העיצוב או בזמן ריצה.
- להירשם לאירועים הנחשפים על ידי Windows Form והפקדים, ולעבד את אותם אירועים.

לאחר סיום כל התרגילים ועיון בכל הדוגמאות שבשלושת חלקיו הראשונים של הספר, עליך כבר להיות מיומן במידה מספקת בשפת Visual C#. למדת לכתוב תוכנות וליצור רכיבים בעזרת C#, ועליך כבר להבין חלק גדול מהדקויות של השפה, כגון ההבדלים בין סוגי ערך וסוגי הפניה. מכיוון שכבר רכשת את יכולות השפה החיוניות, חלק זה, הרביעי, יסייע לך להרחיב את היכולות הללו ולהשתמש ב-C# כדי לנצל את ספריות ממשק המשתמש הגרפי – GUI (Graphical User Interface) הכלולות ב- NET Framework. עיקר הלמידה תתרכז באופן השימוש באובייקטים ממרחב השמות **System.Windows.Forms** ליצירת יישומי Windows Forms.

בפרק זה תלמד לבנות יישום Windows Forms בסיסי באמצעות הרכיבים השכיחים, המהווים חלק ממרבית יישומי GUI. תלמד לשנות את המאפיינים של Windows Forms ושל הפקדים באמצעות **מעצב התצוגה** (Visual Designer) וחלונות Properties. תלמד גם לבדוק ולשנות את הערכים של מאפיינים אלה באופן דינמי על ידי קוד C#. לבסוף תלמד לתפוס ולטפל במספר אירועים שכיחים הנחשפים על ידי Windows Forms.

יצירת יישומים משלך

בתור דוגמה, נניח שעליך ליצור יישום המאפשר למשתמש להזין ולהציג את פרטי החברים בארגון "מצלצלי הפעמונים של מידלשייר", המרכז חבורה מובחרת של יצרני פעמונים. בהתחלה היישום יהיה פשוט ביותר, ותצטרך להתרכז בעיקר בעיצוב הטופס ולוודא שהכל פועל כשורה. אחר כך תוסיף פריטים ליישום ותלמד לבצע בדיקות תקינות (validation tests) כדי לוודא שהנתונים המוזנים אינם חסרי-משמעות. הבט בתמונה הבאה כדי לקבל מושג כיצד היישום ייראה לאחר שתסיים אותו.



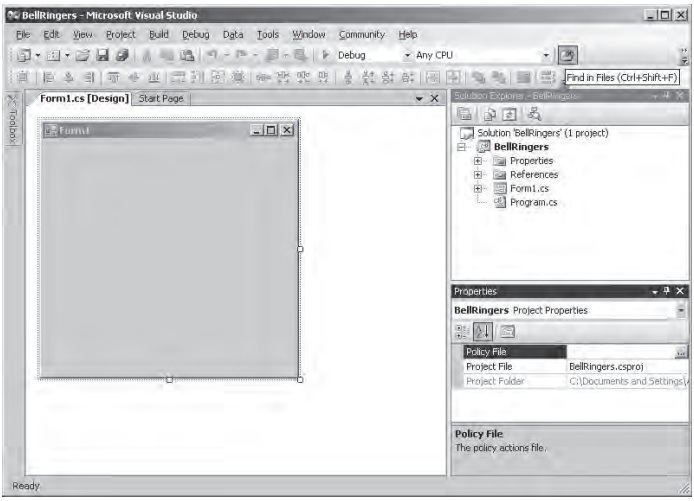
יצירת יישום Windows Forms

בתרגיל הבא עליך להתחיל בבניית היישום של ארגון מצלצלי הפעמונים על ידי יצירת פרויקט חדש, עיצוב הטופס והוספת פקדים (controls) של Windows Forms לטופס. מכיוון שבפרקים הקודמים כבר השתמשת ביישומי Windows Forms בסביבת Microsoft Visual Studio 2005, חלק גדול משני התרגילים הראשונים יהיה מעין חזרה.

צור את פרויקט ארגון מצלצלי הפעמונים של מידלשייר

1. הפעל את Microsoft Visual Studio 2005.
2. בתפריט File, New בחר Project.
3. בחלונית Project Types בחר Visual C#.
4. בחלונית Templates בחר Windows Application.
5. בתיבת הטקסט Name הקלד **BellRingers**.
6. בתיבת הרשימה Location מצא את התיקייה My Documents\Microsoft Press\Visual CSharp Step by Step\Chapter 20.
7. לחץ OK.

כעת ייווצר פרויקט חדש המכיל טופס ריק.



קבע את מאפייני הטופס

1. בחר את הטופס בחלון **תצוגת העיצוב** (Designer View). בחלון Properties, לחץ על המאפיין (Name), ולאחר מכן הקלד **MemberForm** בתיבת הטקסט (Name) כדי לשנות את שם הטופס. אם החלון המתאים אינו מוצג, לחץ בתפריט View על Properties Window, או לחילופין הקש F4.

2. בחלון Properties לחץ על המאפיין **Text**, ולאחר מכן הקלד **Middleshire Bell Ringers Association - Members**, כדי לשנות את הכיתוב שבכותרת הטופס.

3. בחלון Properties, לחץ על המאפיין **BackgroundImage**, ואחר כך לחץ על הלחצן האליפטי שבתיבת הטקסט הסמוכה.

   כתוצאה תיפתח תיבת הדו-שיח Select Resource לבחירת משאבים.

4. בתיבת הדו-שיח Select Resource, בחר Local resource ואחר כך לחץ Import.

   כתוצאה תיפתח תיבת הדו-שיח Open.

5. בתיבת הדו-שיח Open הכנס לתיקייה My Documents\Microsoft Press\Visual CSharp Step by Step\Chapter 20, בחר בקובץ Bell.gif ולחץ Open.

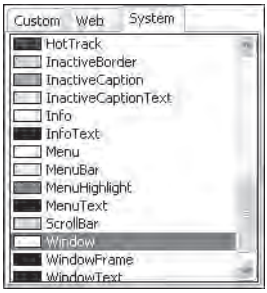
   חלק מהתמונה יוצג בתיבת הדו-שיח Select Resource.

6. בתיבת הדו-שיח Select Resource לחץ OK.

   המאפיין **BackgroundImage** מקובע כעת על התמונה Bell.gif.

7. בחלון Properties, בחר במאפיין **BackColor** ואחר כך לחץ על החץ הפונה מטה שבתיבת בטקסט הסמוכה.

   הפעולה תביא לפתיחת תיבת דו-שיח.



8. בכרטיסייה System שבתיבת הדו-שיח, לחץ Window. ערך זה גורם לכך שכל הפקדים אשר תציב על הטופס יהיו בצבע של החלון עצמו.

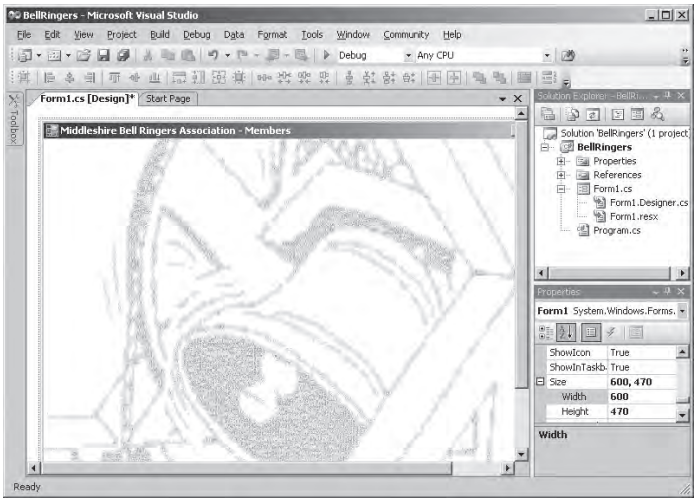
9. בחר במאפיין **Font**, אשר מורכב מסעיפים רבים. בחלון Properties לחץ על סימן הפלוס (+) כדי להרחיב את המאפיין **Font** ולהציג את התכונות השונות. בתכונה **Size** הקלד **12**, ובתכונה **Bold** בחר **True**.

טיפ

חלק מהמאפיינים המורכבים, כגון **Font**, ניתן לשנות באמצעות לחיצה על הלחצן האליפטי שמופיע לאחר בחירה במאפיין. לחיצה על הלחצן האליפטי במאפיין **Font** תגרום להופעת תיבת הדו-שיח הסטנדרטית Font, שבה ניתן לבחור בגופן ובאפקטים הרצויים.

10. שנה את המאפיין **Size** של הטופס (שגם הוא מאפיין מורכב). בחלון Properties, לחץ על סימן הפלוס (+) כדי להרחיב את המאפיין **Size** ולהציג את תכונותיו השונות. את התכונה Width שנה ל-600, ואת התכונה Height שנה ל-470.

הטופס נראה כעת כך:



11. בתפריט Build בחר Build Solution.

וכעת הטופס יוצג ללא תקלות.

12. בתפריט Debug בחר Start Without Debugging.

היישום יופעל ויציג את הטופס הראשי שמכיל את התמונה. הטופס עדיין אינו מבצע פעולה שימושית כלשהי, ולכן סגור אותו וחזור אלVisual Studio.

כיצד פועל יישום Windows Form

ניתן לבנות יישום Windows Form ממספר בלתי מוגבל של טפסים - באפשרותך להוסיף טפסים נוספים ליישום באמצעות הפקודה Add Windows Form שבתפריט Project של סביבת הפיתוח. אם כך, כיצד היישום יודע איזה טופס עליו להציג בעת הפעלת היישום?

אם תתבונן ב- Solution Explorer תראה את הקובץ Program.cs. לחץ עליו לחיצה ימנית ומתפריט הקיצור בחר View Code. תכולת הקובץ תוצג בחלון Code and Text Editor. הקובץ מכיל את השיטה **Main**, המגדירה את נקודת הכניסה של היישום. משפט המפתח של השיטה הוא:

```
Application.Run(new MemberForm());
```

משפט זה יוצר מופע חדש של הטופס **MemberForm** ומציג אותו. כאשר הטופס נסגר, המשפט **Application.Run** נפסק, ומכיוון שזהו המשפט האחרון של **Main**, התוכנית מסתיימת.

טיפ

אם פיתחת בעבר יישומים באמצעות Visual Basic 6, ידוע לך בוודאי שניתן היה להגדיר טופס בתור טופס כברירת מחדל; טופס זה מוגדר באופן אוטומטי בעת הפעלת היישום. אפשרות זו אינה קיימת ב- Visual Studio 2005. אם ברצונך לשנות את הטופס המוצג על ידי יישומי NET Framework בעת הפעלת היישום, עליך לערוך את המשפט **Application.Run** בשיטה **Main**.

ניתן להשתמש במשפט **Application.Run** רק עבור הצגת הטופס הראשוני של היישום. אם הגדרת טפסים נוספים קודם לכן, תוכל להציגם בעזרת קוד משלך, בעיקר במסגרת **מטפל האירועים** (event handler), באמצעות השיטה **Show** שיורשים כל האובייקטים של Windows Form. לדוגמה, אם הוספת ליישום שלך טופס בשם **AnotherForm**, תוכל להציגו באופן הבא:

```
AnotherForm aForm = new AnotherForm();
aForm.Show();
```

מהם מאפייני Windows Form המשותפים?

אם תביט מקרוב על החלון Properties כאשר טופס כלשהו מסומן, תראה שיש יותר מחמישים מאפיינים שונים. חלקם ברורים למדיי, כמו המאפיין Text שאחראי לטקסט המופיע בכותרת. יש מאפיינים שימושיים רק בנסיבות מסוימות. למשל, ניתן להסיר את הלחצנים Minimize, Maximize ו-Close; ניתן גם להסיר את תפריט המערכת מכותרת החלון על ידי שינוי הגדרת המאפיין **ControlBox** ל-**False** - אפשרות שימושית כדי לוודא שהמשתמש לא יסגור את הטופס לפני שיפעיל קוד כלשהו, או שלא יסגור אותו באופן מפורש. מאפיינים אחרים תקפים רק למצבים מסוימים, כמו לדוגמה **Opacity** שיכול לשמש לשליטה ברמת השקיפות של הטופס. הטבלה הבאה מתארת חלק ממאפייני הטפסים המשותפים שניתן לשנות בזמן העיצוב. חשוב לדעת שיש נוספים שאינם מופיעים ב-Properties וניתן לערוך אותם רק על ידי תכנות בסביבת ההרצה. לדוגמה, המאפיין **ActiveControl** מראה איזה מהפקדים בטופס נמצא בשימוש בכל זמן נתון.

מאפיין	תיאור
(Name)	שם הטופס. שני טפסים באותו הפרויקט אינם יכולים להיות בעלי שם זהה.
BackColor	צבע ברירת המחדל לרקע של הטקסט והגרפיקה שבטופס.
BackgroundImage	Bitmap (מפת סיביות), icon (סמל) או כל גרפיקה אחרת המשמשת כרקע של הטופס. אם התמונה קטנה מהטופס, ניתן לרצף, למתוח, למרכז או להגדיל את התמונה באמצעות המאפיין **BackgroundImageLayout**.
Font	גופן ברירת המחדל המשמש את הפקדים שבטופס להצגת הטקסט. זהו מאפיין מורכב. באפשרותך לקבע מספר תכונות שונות שלו וביניהן שם הגופן, גודל, נטייה אלכסונית (italic), מודגש או עם קו-תחתי.
ForeColor	צבע ברירת המחדל של קידמת הטקסטים והגרפיקות שבטופס.
FormBorderStyle	מאפיין זה שולט בסוג ובתצורה של קו המתאר של הטופס. ברירת המחדל היא Sizable. אפשרויות אחרות כוללות קווי מתאר אשר לא ניתן לשנות את גודלם, וכאלה שאינם כוללים את לחצני תפריט המערכת למיניהם.
Icon	מאפיין זה מציין את הסמל שיופיע בתפריט המערכת של הטופס ובשורת המשימות של Microsoft Windows. ניתן ליצור סמלים חדשים באמצעות סביבת הפיתוח Visual Studio 2005.
Location	זהו מאפיין מורכב נוסף המציין את המיקום, או הקואורדינטות, של הפינה השמאלית עליונה של הטופס ביחס לאובייקט המכיל אותו, אשר עשוי להיות טופס אחר או המסך עצמו.
MaximizeBox	מאפיין זה קובע האם תופעל או לא תופעל הפקודה **Maximize** בתפריט המערכת ובכותרת הטופס. על פי ברירת המחדל היא מופעלת.
MaximumSize	מאפיין זה מציין את הגודל המרבי של הטופס. משמעות ערך ברירת המחדל (0, 0) היא, שאין גודל מרבי מותר והמשתמש יכול לשנות את גודל הטופס כרצונו.
MinimizeBox	מאפיין דומה ל-**MaximizeBox**, אשר קובע אם תופעל או לא הפקודה Minimize בתפריט המערכת ובכותרת הטופס. כברירת מחדל היא מופעלת.
MinimumSize	מאפיין זה מציין את הגודל המזערי של הטופס.

מאפיין	תיאור
Size	זהו גודל ברירת המחדל של הטופס כאשר הוא מוצג לראשונה.
Text	מאפיין זה מכיל את הכיתוב שמוצג בשורת הכותרת של הטופס.
WindowState	מאפיין זה קובע את המצב ההתחלתי של הטופס כאשר הוא מוצג לראשונה. מצב ברירת המחדל (Normal) מציב את הטופס על פי המאפיינים **Location** ו-**Size**. האפשרויות האחרות הן Minimized ו-Maximized.

טיפ

כדי לקרוא תיאור על מאפיין כלשהו, בחר במאפיין הרצוי שבחלון Properties, לחץ עליו לחיצה ימנית, ומתפריט הקיצור בחר Description. חלונית המציגה תיאור של המאפיין הנבחר תופיע בתחתית החלון Properties. לחץ על Description בשנית כדי להסתיר את החלונית.

שינוי מאפיינים באמצעות תכנות

בנוסף לעריכת המאפיינים באופן סטטי בזמן העיצוב, ניתן גם לכתוב קוד המשנה את המאפיינים באופן דינמי בסביבת ההרצה. לדוגמה, ניתן לשנות את המאפיין **size** של הטופס כדי לגרום לו לגדול ולהצטמק מבלי שהמשתמש יצטרך לגרור את המסגרת שלו. למעשה, אם תתבונן בקוד שמאחורי הטופס, תראה שסביבת הפיתוח מחוללת קוד אשר משנה את מאפייני הטופס בסביבת ההרצה על פי הערכים שהזנת בזמן העיצוב. אם תלחץ על סימן הפלוס (+) הסמוך ל-Form.cs שב- Solution Explorer, תוכל לראות את הקובץ Form1.Designers.cs; תוכל לראות גם את Form1.resx אשר מכיל מידע על משאבים, כגון bitmaps, המשמשים את היישום שלך. לחץ לחיצה ימנית על הקובץ Form1.Designers.cs ומתפריט הקיצור בחר View Code כדי לצפות בקוד שנוצר. ראית כבר את הקוד הזה בפרק 1, אולם כעת אתה יכול להבין מה הוא עושה.

בקוד זה ניתן לראות שהטופס הוא למעשה מחלקה שיש בה משתנה **System.ComponentModel.IContainer** פרטי בשם **components** וגם השיטה **Dispose**. הממשק **IContainer** מכיל אוסף (collection) שמטרתו לאחסן הפניות למרכיבים השייכים לטופס. השיטה Dispose מממשת את דפוס פינוי הזיכרון שהוסבר בפרק 13, כדי לשחרר במהירות משאבים שנזנחו על ידי הטופס כאשר נסגר.

אם נרחיב את מתחם הקוד שחולל Windows Form Designer נמצא את השיטה InitializeComponent. אם נרחיב אותה נראה כיצד ערכי המאפיינים שהזנת בחלון Properties מתורגמים לקוד. מאוחר יותר, כאשר תוסיף לטופס פקדים, יתווסף לשיטה זו קוד כדי ליצור ולהגדיר אותם. כל שינוי שתבצע בחלון Windows יבוא לידי ביטוי בקוד שבשיטה זו.

חשוב

אל תשנה את הקוד שבשיטה **InitializeComponent** או בכל מקום אחר במתחם הקוד שמחולל Windows Form Designer. כל שינוי שתבצע יאבד ככל הנראה בפעם הבאה שאחד מערכי המאפיינים יעבור שינוי בתצוגת העיצוב.

עליך לשים לב שהקוד שבקובץ Form1.Designers.cs הוא למעשה מחלקה חלקית המשמשת להפרדת המשפטים והשיטות שחוללה סביבת הפיתוח מהקוד שכתבת בעצמך. ב- Solution Explorer, לחץ לחיצה ימנית על Form1.cs ובתפריט הקיצור בחר View Code כדי להציג את הקובץ שנמצאים בו המשפטים והשיטות שכתבת. ניתן לראות שקובץ זה כבר מכיל בנאי ברירת מחדל שתפקידו לקרוא לשיטה **InitializeComponent** כדי שתיצור ותעצב את הטופס בסביבת ההרצה.

הוספת פקדים לטופס

עד כה יצרת טופס, קבעת חלק ממאפייניו, ועיינת בקוד שחוללה סביבת הפיתוח Visual Studio 2005. כדי להפוך את הטופס לשימושי, עליך להוסיף לו פקדים ולכתוב קוד משלך. התיקייה של Windows Form כוללת מגוון פקדים. אשר היעוד של חלק מהם ברור למדיי, כמו לדוגמה **TextBox**, **ListBox**, **CheckBox** ו-**ComboBox**. פקדים אחרים, בעלי עוצמה רבה יותר, כגון **DateTimePicker**, אינם מוכרים כל כך.

שימוש בפקדים של Windows Forms

בתרגיל הבא עליך להוסיף לטופס פקדים שיאפשרו למשתמש להזין פרטים של חבר בארגון. תצטרך להשתמש במספר פקדים שונים שכל אחד מהם מתאים להזנת סוג שונה של נתונים.

תצטרך להשתמש בפקדי **TextBox** (תיבת טקסט) כדי להזין את השם הפרטי ושם המשפחה של החבר. כל חבר שייך ל"מגדל" (שבו תלויים הפעמונים). במחוז מידלשייר יש מספר מגדלים, אבל רשימה זו קבועה, כי לא נבנים מגדלים חדשים בתכיפות גדולה ומגדלים ישנים אינם מתמוטטים או נהרסים. הפקד האידיאלי לסוג נתונים כזה הוא **ComboBox** (תיבת נתונים נפרשת). בעת הזנת חבר חדש יש לציין אם הוא "קפטן" (אחראי מגדל). פקד **CheckBox** (תיבת סימון) הוא ככל הנראה הדרך הטובה ביותר לשלוט בנתון זה, כי ניתן לסמן (**True**) או לבטל את הסימון (**False**).

טיפ

אם המאפיין **ThreeState** נקבע ל-True, לתיבת הסימון **CheckBox** יהיו שלושה מצבים במקום שניים. המצבים הם **True**, **False** ו-**Indeterminate** (לא קבוע). אפשרות זו שימושית כאשר עליך להציג מידע שהתקבל ממאגר נתונים. עמודות מסוימות במאגר נתונים מאפשרים ערכי **null**, המציינות שהערך אינו ידוע או אינו מוגדר. אם ברצונך להציג מידע כזה באמצעות תיבת סימון, תוכל להשתמש במצב **Indeterminate** לייצוג ערכי **null**.

היישום גם אוסף מידע סטטיסטי על תאריך הצטרפות החברים לארגון ועל הניסיון שרכשו בצלצול בפעמונים (עד שנה, בין שנה לארבע שנים, בין חמש לתשע שנים או עשר שנים ומעלה). הפקד **DateTimePicker** מתאים ביותר לבחירה והצגה של תאריכים וקבוצת אפשרויות, או לחצני אפשרויות (radio buttons). כל אלה שימושיים ביותר לציון רמת הניסיון של החבר, כי לחצני אפשרויות מייצגים קבוצת ערכים שאינם יכולים לחפוף.

לבסוף, היישום רושם מידע על הצלילים שהחבר יכול להפיק, אשר נקראים בשם "שיטות" על ידי אגודת מצלצלי הפעמונים. למרות שמצלצל בפעמון יכול להפיק צליל אחד בכל פעם, קבוצה של מצלצלים שבהנהגת הקפטן, יכולה לצלצל ברצפים שונים וכך להפיק מנגינה פשוטה. יש מגוון רב של דרכים לצלצול בפעמון, ויש להן שמות כגון Canterbury Minimus, Plain Bob Doubles ו–Old Oxford Delight Minor. שיטות צלצול חדשות מופיעות כמו פטריות לאחר הגשם, ולכן רשימה זו עשויה להשתנות משנה לשנה. ביישום אמיתי, היית צריך לשמור את רשימת השיטות במאגר נתונים. ביישום זה תשתמש במדגם קטן של שיטות שתטמיע בטופס עצמו (על שימוש במאגרי נתונים תלמד בחלק הבא של הספר). פקד טוב להצגה של מידע מסוג זה ולציון השיטות שהחבר יודע לצלצל הוא **CheckedListBox** (רשימת סימון).

לאחר הזנת כל פרטי החבר, תוכל ללחוץ על הלחצן Add כדי לשמור את הנתונים. המשתמש יכול גם ללחוץ Clear כדי לאפס את הפקדים שבטופס ולבטל את הנתונים שכבר הוזנו.

הוסף פקדי Windows Forms

1. ודא שהטופס Form1 מוצג בחלון תצוגת העיצוב. בערכת הכלים, ודא שהקטגוריה Common Controls פתוחה, ואחר כך גרור את הפקד **Label** ל–**MemberForm**. אם ערכת הכלים אינה מוצגת בתפריט View, בחר Toolbox או לחץ על הכרטיסייה Toolbox שבצידו השמאלי של Visual Studio, או לחץ Ctrl + Alt + X.
2. בחלון Properties בחר במאפיין **Location**, והקלד **10,40** לקביעת מאפיין **המיקום** של התווית.
3. בערכת הכלים, גרור פקד **TextBox** לטופס **MemberForm** ומקם אותו ליד התווית. אל תדאג ליישור מדויק של תיבת הטקסט, מכיוון שנעשה זאת דרך המאפיין **Location**.

טיפ

באפשרותך להשתמש בקווי היישור שבתצוגת העיצוב, כדי ליישר את הפקדים.

4. הוסף לטופס פקד **Label**. מקם אותו מימין לתיבת הטקסט.
5. הוסף פקד **TextBox** טופס **MemberForm** והצב אותו מימין לתווית השנייה.
6. גרור פקד **Label** שלישי לטופס מערכת הכלים. מקם את התווית החדשה מתחת לתווית הראשונה.
7. גרור מערכת הכלים פקד **ComboBox** לטופס. הצב אותו תחת **תיבת הטקסט** הראשונה ומימין ל**תווית** השלישית שב–MemberForm.
8. גרור מערכת הכלים את פקד **CheckBox** אל הטופס. הצב אותו תחת **תיבת הטקסט** השנייה.
9. הוסף תווית (**Label**) רביעית לטופס והצב אותה תחת התווית השלישית.
10. גרור מערכת הכלים פקד **DataTimePicker** לטופס. הצב אותו תחת **התיבה המשולבת**.
11. בערכת הכלים, הרחב את הקטגוריה Containers. גרור פקד **GroupBox** מערכת הכלים ומקם אותו מתחת לפקד **Label** הרביעי.

12. בקטגוריה Common Controls אשר בערכת הכלים, גרור את הפקד **RadioButton** והצב אותו בפקד **GroupBox** שזה עתה הוספת.
13. הוסף שלושה פקדי **RadioButton** אל **GroupBox**, והצב אותם בקו אנכי ישר. ייתכן שתצטרך להגדיל את התיבה **GroupBox** כדי שתוכל להכיל את כולם.
14. גרור מערכת הכלים פקד **CheckedListBox** אל הטופס, והצב אותו תחת התווית השנייה ומימין לפקד **GroupBox**.
15. גרור מערכת הכלים פקד **Button** והצב אותו באזור הפינה השמאלית התחתונה של הטופס **MemberForm**.
16. הוסף לחצן בתחתית הטופס, והצב אותו מימין ללחצן הראשון.

קביעת מאפייני הפקדים

כעת עליך לקבוע את מאפייני הפקדים שהוספת זה עתה לטופס. כדי לשנות את ערך אחד הפקדים, עליך ללחוץ על הבקר בטופס כדי לסמן אותו, ואחר כך להזין ערך למאפיין הרלוונטי בחלון Properties. נתחיל עם המאפיינים הבסיסיים. הטבלה הבאה מכילה את המאפיינים והערכים שעליך להציב עבור כל אחד מהפקדים.

Value	Property	Control
Firsy name	Text	Label1
10, 40	Location	
firstName	(Name)	textBox1
120, 40	Location	
170, 26	Size	
LastName	Text	Label2
300, 40	Location	
lastName	(Name)	textBox2
400, 40	Location	
170, 26	Size	
Tower	Text	label3
10, 92	Location	
towerNames	(Name)	comboBox1
DropDownStyle (קביעה זו מחייבת את המשתמשים לבחור באחד מהפריטים שברשימה; המשתמשים אינם יכולים להקליד בעצמם ערך אחר רצוי).	DropDownStyle	
120, 92	Location	
260, 28	Size	

Value	Property	Control
isCaptain	(Name)	CheckBox1
400, 92	Location	
Captain	Text	
MiddleLeft (מאפיין זה מציין את המיקום של תיבת הסימון ביחס לטקסט שבפקד. כאשר תקיש על החץ למטה של מאפיין זה, תוצג גרפיקה מעניינת המכילה grid. הקש על המשבצת השמאלית שבשורה האמצעית).	CheckAlign	
Member Since (טקסט זה צריך להיות מפוצל לשתי שורות. תוכל להקיש על החץ למטה של מאפיין זה כדי להציג עורך טקסט פשוט שגם מאפשר להזין ערכי טקסט בשורות אחדות).	Text	label4
10, 148	Location	
memberSince	(Name)	DateTimePicker
120, 148	Location	
290, 26	Size	
yearExperience	(Name)	groupBox1
10, 204	Location	
260, 160	Size	
Experience	Text	
novice	(Name)	radioButton1
16, 32 (שים לב שמיקום זה הינו יחסי למיכל לחצני הרדיו, **experience GroupBox**).	Location	
Upto 1 year	Text	
intermediate	(Name)	radioButton2
16, 64	Location	
1 to 4 year	Text	
experienced	(Name)	radioButton3
16, 96	Location	
5 to 9 year	Text	
accomplished	(Name)	radioButton4
16, 128	Location	
10 or more years	Text	

Value	Property	Control
methods	(Name)	checkedListBox1
300, 212	Location	
270, 165	Size	
True	Sorted	
Add	(Name)	button1
190, 388	Location	
75, 40	Size	
Add	Text	
Clear	(Name)	button2
335, 388	Location	
75, 40	Size	
Clear	Text	

בשלב זה מומלץ לשמור את הפרויקט.

מאפייני הפקדים

כפי שזה עתה ראית, בדומה לטפסים גם לפקדים יש מאפיינים רבים שביכולתך לשנות. לכל סוג פקד יש מאפיינים שונים. כמו כן, בדומה לטפסים תוכל לשנות ולבדוק את מאפייני הפקדים באופן דינמי בתוכניות שאתה כותב. יש מספר מאפיינים שזמינים בסביבת ההרצה בלבד. מידע נוסף על המאפיינים השונים של הפקדים ניתן למצוא ב- MSDN Library for Visual Studio 2005 הכלולה ב- Visual Studio 2005.

שינוי מאפיינים באופן דינמי

עד כה השתמשת בתצוגת העיצוב כדי לשנות מאפיינים בצורה סטטית. כאשר הטופס פועל, רצוי לאפס את ערכי הפקדים השונים לערך ברירת המחדל ההתחלתי שלהם. כדי לעשות זאת, עליך לכתוב קוד. בתרגילים הבאים, תיצור שיטה פרטית **Reset**. אחר כך תפעיל את השיטה הזו בעת הפעלת הטופס, וכאשר המשתמש לוחץ על הלחצן Clear.

במקום לכתוב את הקוד מאפס, תוכל להשתמש בעורך Class Diagram כדי לחולל את השיטה. עורך זה מאפשר לצפות במחלקות ולערוך אותן בצורה סכמתית.

צור את השיטה Reset

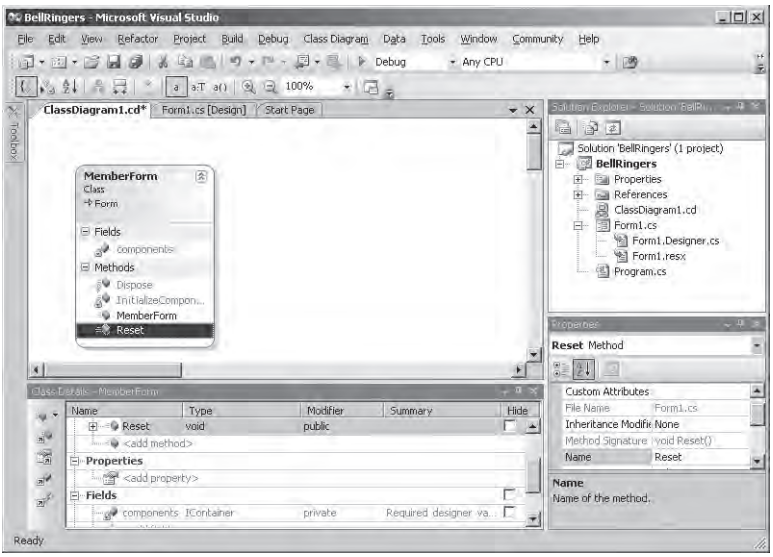
1. ב- Solution Explorer, לחץ לחיצה ימנית על Form1.cs וכתוצאה יופיע תפריט קיצור.
2. בתפריט בחר View Class Diagram.

   על המסך תוצג דיאגרמת מחלקה חדשה המציגה את המחלקה MemberForm.
3. לחץ לחיצה ימנית על המחלקה MemberForm שבדיאגרמה, בתפריט הקיצור הצבע על Add ובחר Method. המחלקה MemberForm תתרחב ותציג את רשימת כל השיטות המוגדרות (**Dispose**, **InitializeComponent** ואת הבנאי **MemberForm**). לרשימה תיווסף עוד שיטה בשם **Method**. שנה את שם השיטה ל-Reset באמצעות הקלדת השם על השם הישן. לסיום הקש Enter.
4. בחלונית Class Details שתחת דיאגרמת המחלקה, ודא שסוג המחלקה Reset הוא **void**, ושבקר הגישה הוא **public**. אם אין זה כך, תוכל לתקן זאת דרך אותם השדות בחלונית Class Details.

טיפ

החלונית Class Details שותפה לעיתים בחלון עם החלונית Error List. אם אינך רואה את החלונית Class Details, לחץ על הכרטיסייה בשם זה שנמצאת תחת חלונית Error List כדי להציגה.

במסך הבא ניתן לראות את דיאגרמת המחלקה הכוללת את השיטה החדשה.



5. בחלונית Class Details שתחת דיאגרמת המחלקה, לחץ לחיצה ימנית על השיטה Reset ומתפריט הקיצור בחר View Code.

כתוצאה תמצא את עצמך בחלון Code and Text Editor אשר מציג את המחלקה MemberForm. השיטה Reset נוספה למחלקה ומכילה מימוש ברירת מחדל לזריקת חריג **NotImplementedException**:

```
public void Reset()
{
   throw new System.NotImplementedException();
}
```

6. בחלון Code and Text Editor, החלף את המשפט **throw** שבשיטה **Reset** עם שתי שורות הקוד הבאות:

```
firstName.Text = “”;
lastName.Text = “”;
```

שני המשפטים הללו מציבים מחרוזת ריקה במאפיין **Text** של תיבות הטקסט **firstName** ו-**lastName** כדי לוודא שהן תהיינה ריקות.

תכנות ממשק המשתמש

כעת עליך לערוך את מאפייני הפקדים האחרים שבטופס באמצעות תכנות.

הוסף ערכים לתיבה המשולבת (ComboBox). התיבה המשולבת **towerName** אמורה להכיל רשימה של כל מגדלי הפעמונים במחוז מידלשייר. מידע זה מאוחסן בדרך-כלל במאגר נתונים, ועליך לכתוב קוד כדי לשלוף את רשימת המגדלים מהמאגר ולמלא באמצעותה את **התיבה המשולבת**. מכיוון שאינך יודע עדיין כיצד לעבוד עם מאגרי נתונים, נשתמש ביישום זה באוסף שמוטבע בקוד.

לתיבה המשולבת יש מאפיין **Items** המכיל רשימה של נתונים שיש להציג. בשיטה **Reset**, תחת הקוד שכבר כתבת, הוסף את המשפטים הבאים שמטרתם לאפס את הרשימה (חשוב לעשות זאת כדי למנוע כפילויות של ערכים ברשימה), ולהוסיף ארבעה פריטים חדשים **לתיבה המשולבת**:

```
towerNames.Items.Clear();
towerNames.Items.Add(“Great Shevington”);
towerNames.Items.Add(“Little Mudford”);
towerNames.Items.Add(“Upper Gumtree”);
towerNames.Items.Add(“Downley Hatch”);
```

הגדר את התאריך הנוכחי. השלב הבא הוא אתחול הבקר memberSince DateTimePicker לתאריך הנוכחי. תוכל לשנות את התאריך באמצעות המאפיין Value. את התאריך הנוכחי תוכל לשלוף מהמאפיין הסטטי Today של המחלקה DateTime. הוסף את המשפט הבא לשיטה Reset:

```
memberSince.Value = DateTime.Today;
```

אתחל את תיבת הסימון (CheckBox). ברירת המחדל של תיבת הסימון **isCaptain** צריכה להיות **False**. כדי לעשות זאת, עליך להגדיר את המאפיין **Checked**. הוסף את המשפט הבא לשיטה **Reset**:

```
isCaptain.Checked = false;
```

אתחל את קבוצת לחצני האפשרויות (radio button group). הטופס מכיל ארבעה לחצני אפשרויות המייצגים את שנות הניסיון של חבר הארגון. לחצן אפשרויות דומה ל**תיבת סימון** מכיוון שהוא יכול להכיל ערכי **True** או **False**. עם זאת, יש אלמנט נוסף כאשר מאגדים מספר לחצני אפשרויות יחדיו בתיבת קבוצה (**GroupBox**). במקרה כזה, ניתן לסמן רק לחצן אחד מבין הלחצנים שבקבוצה (ערך **True**), ואז הסימון של כל הלחצנים האחרים יבוטל (ערך **False**). ברירת המחדל היא שאף אחד מהלחצנים אינו מסומן. תוכל לשנות זאת באמצעות הגדרת המאפיין **Checked** של לחצן האפשרויות **novice**. הוסף את המשפט הבא לשיטה **Reset**:

```
novice.Checked = true;
```

הוסף ערכים לרשימת הסימון (CheckedListBox). בדומה לתיבה המשולבת **Tower**, רשימת הסימון המכילה את רשימת השיטות השונות לצלצול בפעמונים כוללת מאפיין בשם **Items** המכיל את אוסף (collection) הערכים אותם יש להציג. כמו כן, בדומה לתיבה המשולבת, ניתן להוסיף לה ערכים ממאגר נתונים, אך גם במקרה זה נוסיף את הערכים לקוד באופן ישיר. הוסף את המשפטים הבאים לשיטה **Reset**:

```
methods.Items.Clear();
methods.Items.Add("Canterbury Minimus");
methods.Items.Add("Reverse St Nicholas");
methods.Items.Add("Plain Bob Doubles");
methods.Items.Add("Grandsire Doubles");
methods.Items.Add("Cambridge Minor");
methods.Items.Add("Old Oxford Delight Minor");
methods.Items.Add("Kent Treble Bob Major");
```

קרא לשיטה Reset. עליך להוסיף קריאה לשיטה **Reset** אשר תופעל בעת הפעלת הטופס. מקום טוב למיקום המשפט הוא הבנאי **MemberForm**. בחלון Code and Text Editor, אתר את הבנאי **MemberForm** בתחילת המחלקה **MemberForm** שבקובץ Form1.cs. הוסף קריאה לשיטה **Reset** מייד לאחר המשפט הקורא לשיטה **InitializeComponent**:

```
this.Reset();
```

הדר את היישום ובדוק אותו

1. רצוי לקרוא לקובץ המכיל את הטופס בשם זהה לשם הטופס עצמו. ב- Solution Explorer, לחץ לחיצה ימנית על Form1.cs ובתפריט הקיצור בחר Rename. הקלד את השם **MemberForm.cs**.
2. בתפריט Debug בחר Start Without Debugging כדי לוודא שניתן להדר ולהפעיל את הפרויקט.

3. כאשר הטופס פועל, לחץ על **התיבה המשולבת** Tower.
מהרשימה שתוצג, בחר באחד המגדלים.

4. לחץ על החץ שמימין לבורר השעה והתאריך Member Since.
כעת יוצג לוח שנה, שבו ערך ברירת המחדל הוא התאריך הנוכחי. לחץ על אחד התאריכים, ובחר חודש בעזרת החצים. לחיצה על שם החודש תגרום להצגת כל החודשים ברשימה נפתחת, ולחיצה על השנה תאפשר לך לבחור שנה באמצעות פקד נומרי עולה ויורד.

5. לחץ על כל אחד מלחצני האפשרויות שבקבוצה Experience.
שים לב שאינך יכול לסמן יותר מלחצן אחד במקביל.

6. ברשימת הסימון Methods, לחץ על חלק מהשיטות וסמן את תיבת הסימון שלהן. עליך ללחוץ פעם אחת כדי לבחור בשיטה ופעם נוספת כדי לסמן את התיבה או לבטל את הסימון.

7. סגור את הטופס וחזור אל Visual Studio 2005.

חשיפת אירועים ב- Windows Forms

אם עבדת בעבר עם Microsoft Visual Basic, Microsoft Foundation Classes (MFC) או כל כלי אחר המשמש לבניית יישומי GUI עבור Windows, אתה בוודאי יודע ש-Windows משתמש במודל **שמופעל על ידי אירועים** (event-driven model) כדי לקבוע מתי להפעיל קוד. בפרק 16 למדת להציף אירועים משלך ולהירשם אליהם (subscribe to). טפסי Windows והפקדים השונים כוללים אירועים מוגדרים מראש שניתן לעשות אליהם מנוי, ואשר אמורים לכסות את מרבית המקרים האפשריים.

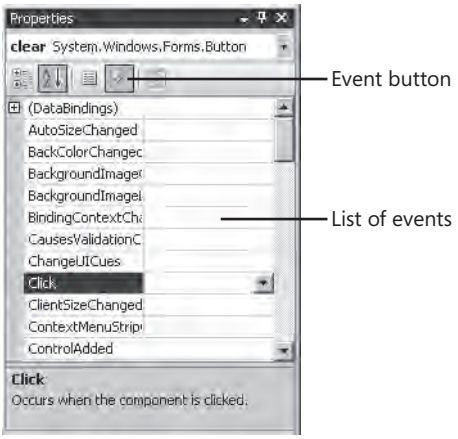
עיבוד אירועים ב- Windows Forms

משימתך כמתכנת היא ללכוד את האירועים שלדעתך רלוונטיים ליישום, ולכתוב קוד אשר מגיב לאירועים הללו. דוגמה נפוצה היא הפקד **Button**, אשר מציף אירוע "מישהו לחץ על הלחצן" כאשר המשתמש לוחץ עליו בעכבר או מקיש Enter כשהלחצן מסומן. אם רצונך שהלחצן יבצע פעולה כלשהי, עליך לכתוב קוד המגיב לאירוע. בתרגיל האחרון בפרק זה תכתוב את הקוד הדרוש.

טפל באירוע Click של הלחצן Clear

1. בתצוגת העיצוב (בתפריט View בחר Designer), סמן את הלחצן Clear בטופס **MemberForm**.
אנו רוצים שלחיצה על הלחצן Clear תגרום להשבת ערכי ברירת המחדל של הטופס.

2. בחלון Properties לחץ Events.



רשימת המאפיינים תוחלף ברשימת אירועים שביכולתך ללכוד.

3. סמן את האירוע **Click**.

4. בתיבת הטקסט הקלד **clearClick**. הקש Enter.

פעולה זו תגרום ליצירת שיטת אירוע חדשה בשם **clearClick**, והצגתה בחלון Code and Text Editor. שים לב שהגדרות שיטת האירוע עולות בקנה אחד עם ההגדרות המקובלות, מכיוון שהיא מקבלת שני פרמטרים: השולח (מסוג **object**) וארגומנטים נוספים (מסוג **EventArgs**). סביבת ההרצה של Windows Form מעבירה לפרמטרים הללו מידע על מקור האירוע וכל מידע נוסף אשר עשוי להיות שימושי בעת הטיפול באירוע. בתרגיל זה אין צורך בפרמטרים אלה.

5. בגוף השיטה **clearClick**, קרא לשיטה **Reset**.

השיטה נראית כעת בדיוק כך:

```
private void clearClick(object sender, System.EventArgs e)
{
   this.Reset();
}
```

טפל באירוע Click של הלחצן Add

המשתמש ילחץ על הלחצן Add כאשר ירצה לאחסן את כל הנתונים של חבר הארגון שהוזנו לטופס. האירוע **Click** צריך לבדוק את כל הנתונים שהוזנו כדי לוודא שאינם חסרי-הגיון (לדוגמה, האם ייתכן שלקפטן של מגדל תהיה פחות משנת ניסיון אחת?). אם הכל תקין, האירוע ישלח את הנתונים למאגר נתונים או לאמצעי אחסון קבוע אחר. על תקינות המידע ואחסון נתונים תלמד בפרקים הבאים. כעת עליך ליצור קוד עבור האירוע **Click** של הלחצן Add, אשר יציג בפני המשתמש הודעה המכילה את הנתונים שהתקבלו.

1. חזור לתצוגת העיצוב ובחר בלחצן Add.

2. ודא שהחלון Properties מציג את האירועים ולא את המאפיינים. באירוע **Click** הקלד **addClick** והקש Enter.

פעולה זו תגרום לבניית שיטת אירוע נוסף בשם **addClick**.

3. הוסף את הקוד הבא לשיטה **addClick**:

```
string details;
details = “Member name “ + firstName.Text + “ “ + lastName.Text
    + “ from the tower at “ + towerNames.Text;
MessageBox.Show(details, “Member Information”);
```

בלוק זה של קוד יוצר משתנה מחרוזת בשם **details** ומציב בו את שם החבר ואת שם המגדל שהוא משויך אליו. שים לב לדרך שבה הקוד ניגש אל המאפיין **Text** של תיבת הטקסט ושל התיבה המשולבת, כדי לקרוא את הערך הנוכחי של פקדים אלה. המחלקה **MessageBox** מכילה שיטה סטטית להצגת תיבות הדו-שיח על המסך. השיטה **Show** משמשת במקרה זה להצגת תכולת המחרוזת **details** בגוף תיבת ההודעה ומציבה את הטקסט "Member Information" בכותרת החלון. **Show** היא שיטה מועמסת, אשר כוללת גרסאות נוספות המאפשרות להוסיף סמלים ולחצנים לתיבת ההודעה.

טפל באירוע Closing של הטופס

1. חזור לתצוגת העיצוב ובחר בטופס. עשה זאת על ידי לחיצה על נקודה כלשהי ברקע של הטופס ולא על אחד מהפקדים.

2. בחלון Properties, ודא שהאירועים מוצגים. הקלד **memberFormClosing** באירוע **FormClosing** והקש Enter.

כתוצאה נוצר אירוע נוסף בשם **memberFormClosing**.

שים לב שהפרמטר השני של שיטה זו הוא מסוג **FormClosingEventArg**. המחלקה **FormClosingEventArg** מכילה מאפיין בוליאני Cancel. אם ערך **Cancel** שבמטפל האירוע הוא true, הטופס לא ייסגר. אם **Cancel** הוא **false** (ערך ברירת המחדל), הטופס ייסגר כאשר המטפל באירוע יסיים את פעולתו.

3. הוסף את המשפטים הבאים לשיטה **memberFormClosing**:

```
DialogResult key = MessageBox.Show(
   “Are you sure you want to quit ?”,
   “Confirm”,
   MessageBoxButtons.YesNo,
   MessageBoxIcon.Question);
e.Cancel = (key == DialogResult.No);
```

משפטים אלה מציגים תיבת הודעה המבקשת מהמשתמש לאמת שהוא אכן רוצה לסגור את היישום. בתיבת ההודעה יהיו שני לחצנים, Yes ו-No, וסמל סימן שאלה (?). כאשר המשתמש לוחץ Yes או No, תיבת ההודעה תיסגר והשיטה תחזיר את ערך הלחצן הנבחר (בתור **DialogResult**, שהינו משתנה רשימה המציין איזה כפתור נלחץ).

אם המשתמש לוחץ No, המשפט השני יציב ערך true במאפיין **Cancel** של הפרמטר **CancelEventArgs**, וכך תימנע סגירת החלון.

נציגים עבור אירועי Windows Forms

כאשר אתה מגדיר שיטת אירוע (ראה פרק 16) באמצעות חלון Properties, סביבת הפיתוח Visual Studio 2005 מחוללת קוד היוצר נציג (delegate) אשר מפנה לשיטה ואשר נרשם (subscribes) לאירוע. אם תעיין בבלוק הקוד המגדיר את הלחצן Clear בשיטה **InitializeComponent** שבקובץ MemberForm.Designer.cs, תבחין במשפט הבא:

```
//
// clear
//
...
this.clear.Click += new System.EventHandler(this.clearClick);
```

משפט זה נועד ליצור נציג **EventHandler** המפנה לשיטה **clearClick**. אחר כך הנציג נקשר לאירוע של הלחצן Clear. בכל פעם שתיצור שיטות אירוע נוספות, סביבת הפיתוח תחולל את הנציגים המתאימים ותשייך אותם לאירועים במקומך.

הפעל את היישום

1. בתפריט Debug בחר Start Without Debugging כדי להפעיל את היישום.
2. הקלד את השם הפרטי ואת שם המשפחה, ואחר כך בחר מגדל מתוך הרשימה. לחץ Add. תיבת ההודעה תציג את פרטי החבר שהזנת. לחץ OK.
3. נסה לסגור את הטופס. בתיבת ההודעה שתופיע, לחץ No.
   הטופס ימשיך לפעול.
4. נסה לסגור את הטופס שוב, אך הפעם לחץ Yes.
   הפעם הטופס ייסגר והיישום יסתיים.

☯ **אם ברצונך להמשיך לפרק הבא**

השאר את Visual Studio 2005 פעילה ועבור לפרק 21.

☯ **אם ברצונך לסגור כעת את Visual Studio 2005**

פתח את תפריט Menu ולחץ Exit. אם מופיעה תיבת דו-שיח Save, לחץ Yes.

פרק 20 – טבלה מסכמת

המשימה	צריך
ליצור פרויקט Windows Form חדש.	להשתמש בתבנית Windows Application.
לשנות את מאפייני הטופס.	ללחוץ על הטופס בתצוגת העיצוב. בחלון Properties לבחור במאפיין שיש לשנות ולהקליד את הערך החדש.
להוסיף פקדים לטופס.	לגרור את הפקד מערכת הכלים לטופס.
לשנות מאפיינים של פקד.	בתצוגת העיצוב, ללחוץ על הפקד. בחלון Properties לבחור במאפיין שיש לשנות ולהקליד ערך חדש.

להוסיף ערכים לתיבה משולבת (ComboBox) או לרשימת סימון (ListBox).	להשתמש בשיטה Add של המאפיין Items. לדוגמה:
	`towerNames.Items.Add("Upper Gumtree");`
	ייתכן שתצטרך לרוקן את הרשימה אם אין צורך לשמור את הערכים הקיימים ברשימה. לדוגמה:
	`towerNames.Items.Clear();`

לטפל באירוע של פקד או טופס.	לבחור את הפקד או הטופס בתצוגת העיצוב. בחלון Properties יש ללחוץ על הלחצן Events. למצוא את האירוע שצריך לטפל בו, ולהקליד את שם שיטת האירוע. כעת ניתן לכתוב את הקוד לטיפול באירוע בגוף שיטת האירוע.

פרק 21

עבודה עם תפריטים ותיבות דו-שיח

כאשר תסיים פרק זה, תוכל:

- ליצור ולערוך תפריטים עבור יישומי Windows Forms.
- להשתמש בפקד MenuStrip ובפקד **ContextMenuStrip**.
- להגיב לאירועי תפריטים שמטרתם ביצוע עיבודים כאשר המשתמש בוחר בפקודה מהתפריט.
- לשלוט בתפריטים באמצעות תכנות וליצור תפריטים דינמיים.
- ליצור תפריטי קיצור המתאימים עצמם לנסיבות.
- להשתמש בתיבות דו-שיח משותפות כדי לאפשר למשתמש לבחור בשמות לקבצים ולבחור מדפסות.
- לשלוח מסמך למדפסת.

בפרק 20 למדת ליצור יישום פשוט של Windows Forms אשר משתמש בפקדים ואירועים שונים. יישומים מקצועיים רבים של Microsoft Windows כוללים גם תפריטים המכילים פקודות ואפשרויות, וכך מאפשרים למשתמש לבצע פעולות רבות במסגרת היישום. בפרק זה תלמד ליצור תפריטים ולהוסיפם לטפסים באמצעות הפקד MenuStrip; תלמד להגיב כאשר המשתמש בוחר בפקודה מהתפריט; תלמד גם ליצור תפריטי קיצור שתוכנם נקבע על פי ההקשר של פתיחתם; ולבסוף תלמד על תיבות הדו-שיח המשותפות הכלולות בספרייה Windows Form. הפקדים הללו מאפשרים למשתמש גישה נוחה וברורה לפריטים שהשימוש בהם שכיח, כגון קבצים ומדפסות.

קווים מנחים ליצירה ועיצוב של תפריטים

ברוב יישומי Windows, יש פריטים מסוימים בשורת התפריטים (menu strip) אשר מוצגים לרוב באותו מקום, ותוכנם בדרך כלל דומה. לדוגמה, תפריט File (קובץ) יהיה לרוב הראשון בשורת התפריטים, ובדרך כלל תוכל למצוא בו את הפקודות המשמשות ליצירת מסמך חדש, פתיחת מסמך קיים, שמירת מסמך, הדפסת מסמך ויציאה מהיישום. **מסמך** (**document**) הוא למעשה הנתונים שהיישום מעבד. ביישום Microsoft Excel, יהיה זה גיליון הנתונים (spreadsheet) וביישום של מצלצלי הפעמונים שייצרת בפרק 20, יהיה זה חבר חדש.

גם סדר הפקודות שבתפריט יהיה לרוב זהה בכל היישומים. לדוגמה, הפקודה Exit (יציאה) תהיה לרוב האחרונה בתפריט File. בנוסף, התפריט עשוי להכיל פקודות ייחודיות לכל יישום.

כל יישום כולל לרוב גם תפריט Edit (עריכה) שמכיל פקודות כגון Cut, Paste, Clear ו-Find. בשורת התפריטים תמצא גם תפריטים ייחודיים לאותו יישום, אך מקובל שהתפריט האחרון יהיה תמיד Help (עזרה) שמכיל גישה לקובצי העזרה של היישום וגם למידע "אודות", אשר מכיל את מספר הגרסה, פרטי זכויות היוצרים ורישיון השימוש ביישום. ביישום מתוכנן היטב, תכולת התפריטים השונים צריכה להיות צפויה ועליהם לאפשר הפעלה פשוטה של היישום על ידי המשתמש הסביר.

טיפ

חברת Microsoft פרסמה קווים מנחים ליצירת ממשק משתמש, הכולל גם עיצוב תפריטים. תוכל למצוא מידע זה באתר של החברה שכתובתו **http://msdn.microsoft.com/ui**.

הוספת תפריטים ועיבוד אירועי תפריטים

התוכנה Visual Studio 2005 של Microsoft מאפשרת למשתמש להוסיף תפריטים ופריטי תפריטים בשתי דרכים שונות. ניתן להשתמש בסביבת הפיתוח המשולבת **IDE** (Integrated Development Environment) של סביבת הפיתוח ובעורך התפריטים כדי ליצור תפריטים בצורה גרפית. ניתן גם לכתוב קוד שיוצר אובייקט **MenuStrip** (זו מחלקה שמוגדרת במסגרת הספרייה Windows Form), ולהוסיף לו אובייקטים מסוג **ToolStripMenuItem** (מחלקה זו מוגדרת גם היא במסגרת הספרייה Windows Form).

עבודתך אינה מסתיימת בעיצוב התפריט. כאשר המשתמש בוחר באחת הפקודות מהתפריט, הוא מצפה שיקרה משהו! כדי להפעיל את הפקודות, עליך ללכוד אירועי תפריט ולהפעיל את הקוד בצורה דומה לאופן הטיפול באירועי הפקדים.

יצירת תפריט

בתרגיל הבא עליך להשתמש ב-IDE כדי ליצור תפריטים עבור היישום של ארגון מצלצלי הפעמונים של מידלשייר. בהמשך הפרק תלמד לבנות ולשלוט בתפריטים על ידי תכנות.

צור את תפריט File

1. הפעל את Visual Studio 2005.
2. פתח את הפרויקט **BellRingers**, שבתיקייה My Documents\Microsoft Press\Visual CSharp Step by Step\Chapter 21\BellRingers. זהו היישום של מצלצלי הפעמונים של מידלשייר, שאמור להיות זהה לגרסה שיצרת בעצמך בפרק 20.

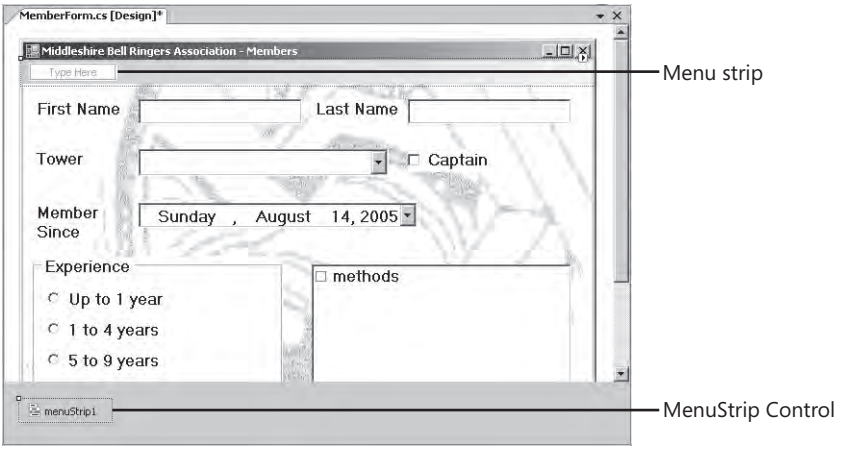
3. הצג את הטופס **MemberForm** בחלון תצוגת העיצוב (Design View). (ב- Solution Explorer לחץ על הקובץ MemberForm.cs, ואחר כך בתפריט View בחר Designer, או לחילופין לחץ לחיצה כפולה על שם הקובץ).

כעת יופיע הקובץ **MemberForm**.

4. בערכת הכלים, הרחב את הקטגוריה Menus & Toolbars. גרור את הבקר **MenuStrip** (שורת תפריטים) אל הטופס **MemberForm**.

חשוב

עליך לשחרר את הפקד MenuStrip על פני הטופס, ולא על אחד הפקדים שבטופס. פקדים מסוימים, כגון GroupBox, יכולים להכיל תפריטים משלהם, ואם תשחרר תפריט על פקד מסוג זה, התפריט יצורף אליו ולא לטופס.



שם ברירת המחדל **menustrip1** של הפקד יופיע בתחתית הטופס, ושורת תפריטים עם הכיתוב Type Here תופיע בחלק העליון של הטופס.

5. לחץ על הכיתוב Type Here שבשורת התפריטים, הקלד **&File** והקש Enter. אם הכיתוב שבשורת התפריטים נעלם, לחץ על הפקד **mainStrip1** שבתחתית הטופס ואז התפריט יופיע שוב.

לאחר שתלחץ על הכיתוב Type Here, כיתוב Type Here נוסף יופיע מימין לפריט הנוכחי, וכיתוב Type Here שלישי יופיע מתחת לתפריט File.

טיפ

הצבת התו & לפני אות יוצרת גישה מהירה לתפריט על ידי הקשת Alt בשילוב האות שאחרי התו & (עבור &File תוכל להקיש Alt+F). זהו קיצור מקובל בכל היישומים. כאשר המשתמש לוחץ Alt, קו תחתי יופיע תחת האות F של File. אל תשתמש באותו מקש גישה יותר מפעם אחת בכל תפריט כי הדבר עלול לבלבל את המשתמש, ויש להניח שגם את היישום עצמו.

6. לחץ על הכיתוב Type Here שתחת התפריט File, הקלד **&New** והקש Enter.
כיתוב Type Here נוסף יופיע מתחת לפריט New.

7. לחץ על הכיתוב Type Here שתחת הפריט New, הקלד **&Open** והקש Enter.

טיפ

כדי לערוך את השם של אחד מפריטי התפריט, לחץ עליו, ובחלון Properties שנה את המאפיין Text שלו.

8. לחץ על הכיתוב Type Here שתחת הפריט Open, הקלד **&Save Member** והקש Enter.

9. לחץ על הכיתוב Type Here שתחת הפריט Save Member, הקלד את הסימן מינוס (-) והקש Enter.
פעולה זו תיצור קו הפרדה שנועד לאגד פריטים בתפריט הקשורים זה לזה.

10. לחץ על הכיתוב Type Here שתחת קו ההפרדה, הקלד **&Print** והקש Enter.

11. לחץ על הכיתוב Type Here שתחת הפריט Print, הקלד את הסימן מינוס (-) והקש Enter.
קו הפרדה נוסף יופיע בתפריט.

12. לחץ על הכיתוב Type Here שתחת קו ההפרדה השני, הקלד **E&xit** והקש Enter.

שים לב לכך שמקש הגישה המקובל ליציאה מהתוכנית הוא "x". בסיום כל השלבים התפריט נראה כך:

סוגי הפריטים בשורת התפריטים

עד כה השתמשת בפקדים מסוג MenuItem, שהוא סוג ברירת המחדל לאובייקטים שבשורת התפריטים. ביישומים של Visual Studio 2005 ניתן להוסיף גם תיבות משולבות ותיבות טקסט. ייתכן ששמת לב לחץ התפריט-הנפתח המופיע כאשר מציבים את סמן העכבר מעל הכיתוב Type Here שבפקד MenuStrip. אם תפתח את התפריט, תבחין בשלושה פריטים: MenuItem, ComboBox, TextBox, כלהלן:

כאשר פריטים אלה מופיעים בשורת התפריטים, התיבה המשולבת ותיבת הטקסט פועלות בצורה דומה לפקדים ComboBox ו-TextBox שבטופס, וניתן לשנות את מאפייניהן כפי שמוצג בפרק 20. עם זאת, תיבות אלו אינן זהות לגמרי לפקדים שמציבים על טפסים. הפקדים ToolStripComboBox ו-ToolStripTextBox מעוצבים במיוחד לשימוש במסגרת פקדים מסוג MenuStrip, או פקדי "strip" אחרים, כגון ContextMenuStrip ו-ToolStrip.

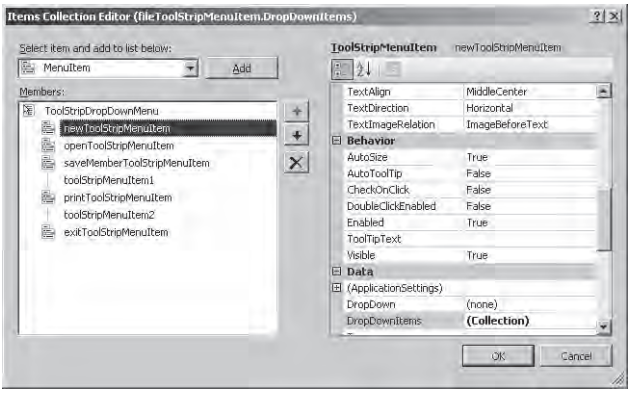
קביעת המאפיינים של פרטי התפריט

בתרגיל הבא עליך לקבוע באמצעות החלון Properties את מאפייני הפקד **MainMenu** ושל פריטי התפריט.

קבע את מאפייני פריטי התפריט

1. לחץ על הפקד **menuStrip1** שמתחת לטופס. בחלון Properties, שנה את שמו ל-**mainMenu** (אם החלון Properties אינו מופיע על המסך, בתפריט View בחר Properties Window או הקש F4).
2. לחץ לחיצה ימנית על התפריט File שבשורת התפריטים של הטופס **MemberForm**. בתפריט הקיצור בחר Edit DropDownItems.

   החלון Items Collection Editor יציג את שמות פריטי התפריט ואת מאפייניהם.



טיפ

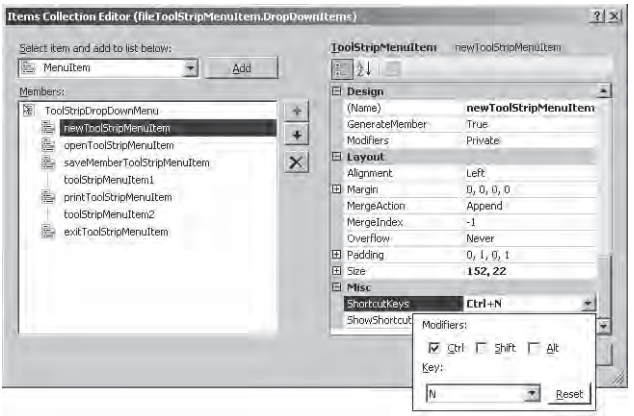
ניתן להשתמש בחלון Items Collection Editor למחיקת פריטי תפריט, להוספת פריטי תפריט חדשים, לשינוי סדר הפריטים בתפריט נפתח ולשינוי מאפייני הפריטים.

3. שים לב לשמות שהוקצו לכל אחד מפריטי התפריט. עליהם להיות זהים לשמות שבטבלה הבאה. אם לא, בחר בפריט המתאים וערוך את המאפיין (Name) שבצד הימני של החלון Items Collection Editor.

Item	New name
New	**newToolStripMenuItem**
Open	**openToolStripMenuItem**
Save Member	**saveMemberToolStripMenuItem**
Print	**printToolStripMenuItem**
Exit	**exitToolStripMenuItem**

במקשי הגישה של פריטי התפריט (כמו למשל Alt+X עבור הפקודה Exit) ניתן להשתמש רק לאחר הצגת התפריט. עבור פריטים שהשימוש בהם נעשה לעיתים תכופות במיוחד, כדאי לקבוע מקשי קיצור שהמשתמש יוכל להקיש בכל עת, כדי להפעיל את הפקודה הרצויה. לדוגמה, ברוב היישומים של Windows אפשר להפעיל את הפקודה New על ידי הקשת Ctrl+N. תוכל להוסיף מקשי קיצור באמצעות המאפיין **ShortcutKeys** של פריט התפריט.

4. בחר מרשימת החברים את **newToolStripMenuItem**, ומרשימת המאפיינים בחר במאפיין ShortcutKeys. סמן את Ctrl Modifier, בחר באות N כמקש קיצור והקש Enter.



5. תחת המאפיין **ShortcutKeys** ישנו המאפיין **ShowShortCut**. הוא נועד לקבוע אם מקש הקיצור יוצג בתפריט בעת הפעלת היישום. ודא שערך המאפיין הוא **True**.

6. שנה את מקשי הקיצור של פריטי התפריט האחרים על פי הערכים שבטבלה הבאה. ודא שהמאפיין **ShowShortCut** של כל אחד מהפריטים מכיל את הערך True.

Item	Shortcut
openToolStripMenuItem	Ctrl+O
saveMemberToolStripMenuItem	Ctrl+S
printToolStripMenuItem	Ctrl+P
exitToolStripMenuItem	Alt+F4

ניתן לחסום (Disable) או לאפשר (Enable) את הפריטים שבתפריט (פריטים חסומים אינם נגישים למשתמש והכיתוב שלהם מעומם) על פי המצב הנוכחי של היישום. לדוגמה, ביישום של ארגון מצלצלי הפעמונים, אין לאפשר למשתמש לשמור את הנתונים של חבר או להדפיסם לפני שהוא מזין נתונים לטופס ולכן פריטי תפריט אלה יהיו חסומים, כלומר עמומים.

7. בחר בפריט **printToolStripMenuItem**, וברשימת המאפיינים לחץ על **Enabled**. בתפריט הנפתח בחר **False**. בהמשך תצטרך לכתוב קוד אשר יעדכן את ערך המאפיין ל-**True** לאחר שהמשתמש מזין מידע לטופס.

   חזור על פעולת החסימה גם עבור הפריט **saveMemberToolStripMenuItem**.

8. לחץ OK כדי לסגור את החלון Items Collection Editor.

בדוק את התפריט

1. בתפריט Debug בחר Start Without Debugging כדי לבצע הידור ולהפעיל את היישום.
2. בטופס שיופיע פתח את תפריט File.

   כתוצאה יופיע התפריט החדש.

שים לב שהפקודות Save ו–Print חסומות (disabled). ניתן לבחור בפקודות אחרות למרות שנכון לעכשיו הן אינן עושות דבר.

3. סגור את הטופס. עדיין לא ניתן להשתמש בפקודה Exit שבתפריט File. לחץ על הלחצן "X" שבפינה הימנית העליונה של הטופס כדי לסגור אותו.

מאפיינים נוספים של פריטי התפריט

לפריטי התפריט יש מאפיינים נוספים, אשר הנפוצים והשימושים ביותר מוצגים בטבלה הבאה. למידע נוסף על מאפיינים מוכרים פחות, עיין בתיעוד של MSDN Library for Visual Studio.

טיפ

כדי לקבע את מאפייני פריטי התפריט ניתן להשתמש בחלון Items Collection Editor באופן שתואר בתרגיל האחרון. לחילופין, ניתן גם לבחור באחד מפריטי התפריט שבחלון תצוגת העיצוב, ולאחר מכן לערוך את מאפייניו בחלון Properties.

המאפיין	תיאור
(Name)	במאפיין זה מאוחסן שם הפריט.
Checked	פריטי תפריט יכולים להתנהג בצורה דומה לתיבות סימון, ואז יופיע סימון ליד הפריט כאשר המשתמש בוחר בו. כאשר הערך של המאפיין **Checked** הוא **True**, יופיע סימון ליד הפריט וכאשר ערכו הופך ל–**False** – הסימון ייעלם.
CheckOnClick	כאשר הערך של המאפיין **CheckOnClick** הוא **True**, הסימון ✓ לצידו של פריט התפריט יופיע וייעלם לסירוגין כאשר המשתמש לוחץ עליו.
DisplayStyle	פריטי התפריט יכולים לכלול גם תמונות. כאשר הערך של המשתנה **DisplayStyle** הוא **Image** או **ImageAndText**, התמונה שבמאפיין **Image** תוצג בתפריט לצד הפריט.
Enabled	מאפיין זה קובע אם הפריט שבתפריט חסום או מאופשר. כאשר פריט התפריט חסום, הכיתוב שלו מעומם והמשתמש אינו יכול ללחוץ עליו.
ShortcutKeys	מאפיין זה מכיל את מקש הקיצור המשמש להפעלה מהירה של פריט התפריט.
ShowShortcutKeys	כאשר הערך של המאפיין הוא **True**, מקש הקיצור מוצג בתפריט לצד הכיתוב של הפריט.
Text	מאפיין זה מכיל את הכיתוב שמייצג את הפריט בתפריט. ניתן לקבוע מקשי גישה על ידי מיקום של התו & לפני האות הרצויה.
ToolTipText	מאפיין זה מכיל את תיבת הטיפ שתוצג כאשר המשתמש מציב את סמן העכבר מעל פריט התפריט.
Visible	מאפיין זה קובע אם הפריט יוצג בתפריט או לא. לרוב נשתמש במאפיין **Enabled** כדי שהמשתמש יידע על קיומו של הפריט גם כשאינו זמין.

אירועי תפריט

אירועים אחדים שימושיים למדיי עשויים להתרחש כאשר המשתמש פועל בשורת התפריטים. האירוע השכיח והשימושי ביותר הוא **Click**, המתרחש כאשר המשתמש לוחץ על אחד מפריטי התפריט. עליך ללכוד את האירוע כדי לבצע את הפעולות המתאימות לפריט.

בתרגיל הבא תלמד על אירועי תפריט ועל הדרכים לעבד אותם. עליך ליצור אירועי **Click** עבור פריטי התפריט New ו-Exit. מטרת הפקודה New לאפשר למשתמש להקליד פרטים של חבר חדש. לכן, השדות בטופס יהיו פעילים רק לאחר שהמשתמש יבחר New. כאשר המשתמש לוחץ New בתפריט File, עליך להפעיל את כל השדות ולאפס את תוכן הטופס **MemberForm** כדי שהמשתמש יוכל להזין בו נתונים חדשים, ולהפעיל את הפקודה Print.

טפל באירועים של פריטי תפריט

1. בטופס **MemberForm** אשר בתצוגת העיצוב, לחץ על תיבת הטקסט **firstName**. בחלון Properties, שנה את ערך המאפיין **Enabled** ל-**False**. חזור על התהליך עבור הפקדים **lastName**, **towerNames**, **isCaptain**, **memberSince**, **yearsExperience**, **methods**, **add** ו-**clear**.

טיפ

אם תלחץ על מספר פקדים בעודך מחזיק את מקש Shift, כולם יסומנו. בדרך זו תוכל לשנות את הערך של אחד המאפיינים, כגון **Enabled**, כדי שיתייחס לכל הפקדים בפעם אחת.

2. פתח את תפריט File ובחר New בתוך הטופס.
3. בחלון Properties לחץ Events. בחר באירוע **Click**, הקלד **newClick** והקש Enter. כתוצאה תיווצר שיטה חדשה וקוד המקור שלה יופיע בחלון Code and Text Editor.
4. בגוף השיטה **newClick**, כתוב את המשפט הבא:

```
this.Reset();
```

משפט זה קורא לשיטה **Reset**, אשר מאפסת את ערכי הפקדים שעל הטופס לערכי ברירת המחדל שלהם. אם אינך זוכר כיצד השיטה פועלת, אתר את השיטה בחלון Code and Text Editor ועיין במידע כדי לרענן את זיכרונך.

5. כעת עליך להפעיל את פריטי התפריט Save Member ו-Print כדי לאפשר למשתמש לשמור ולהדפיס את המידע של חבר הארגון הנוכחי. כדי לעשות זאת, הצב ערך **true** במאפייני **Enabled** של **saveMemberToolStripMenuItem** ושל **printToolStripMenuItem**.

לאחר הקריאה לשיטה Reset שבשיטת האירוע newClick, הוסף את המשפטים הבאים:

```
printToolStripMenuItem.Enabled = true;
saveMemberToolStripMenuItem.Enabled = true;
```

.6 עליך גם להפעיל את הפקדים שעל הטופס. צרף לשיטה **newClick** את המשפטים הבאים:

```
firstName.Enabled = true;
lastName.Enabled = true;
towerNames.Enabled = true;
isCaptain.Enabled = true;
memberSince.Enabled = true;
yearsExperience.Enabled = true;
methods.Enabled = true;
add.Enabled = true;
clear.Enabled = true;
```

.7 כעת עליך ליצור שיטת אירוע Click עבור הפקודה Exit. שיטה זו צריכה לסגור את הטופס. חזור אל הטופס **MemberForm** שבתצוגת העיצוב, ובתפריט File בחר Exit.

.8 ודא שהחלון Properties מציג את רשימת האירועים, ובחר באירוע **Click**. הקלד **exitClick** והקש Enter.

כתוצאה תיווצר שיטת האירוע **ExitClick** וקוד המקור שלה יופיע בחלון Code and Text Editor.

.9 הקלד בגוף השיטה **ExitClick** את המשפט הבא:

```
this.Close();
```

השיטה **Close** של הטופס **תנסה** לסגור את הטופס. לאחר הפעלת הפקודה Exit, המשתמש נשאל אם הוא בטוח שברצונו לצאת מהתוכנית. אם תשובתו שלילית, הטופס **לא ייסגר** והיישום ימשיך בפעולתו.

בדוק את אירועי התפריט

.1 בתפריט Debug בחר Start Without Debugging כדי להדר ולהפעיל את התוכנית.

שים לב שכל השדות בטופס חסומים / עמומים (disabled).

.2 פתח את תפריט File.

הפקודות Print ו–Save חסומות.

.3 לחץ New. פתח את תפריט File בשנית.

כעת הפקודות Print ו–File מופעלות. גם השדות שעל הטופס מופעלים.

4. לחץ Exit, והטופס ינסה להיסגר. בתיבת הדו-שיח תישאל אם אתה בטוח בכוונתך לסגור את הטופס. אם תלחץ No, הטופס יישאר פתוח; אם תלחץ Yes - הטופס ייסגר והיישום יסתיים.
5. לחץ Yes כדי לסגור את הטופס.

תפריטי קיצור

יישומי Windows רבים נעזרים ב**תפריטי קיצור** (**pop-up** menus) שנפתחים בלחיצה ימנית על טופס או פקד. תפריטים אלה **תלויי הקשר** (context-sensitive) והכוונה לכך שהם בדרך כלל מתאימים את עצמם לנסיבות ומציגים פקודות שניתן ליישם רק על הפקד או הטופס הנבחר. תפריטים אלה מכונים לעיתים **תפריטי הקשר** (**context** menus) או **תפריטים קופצים**.

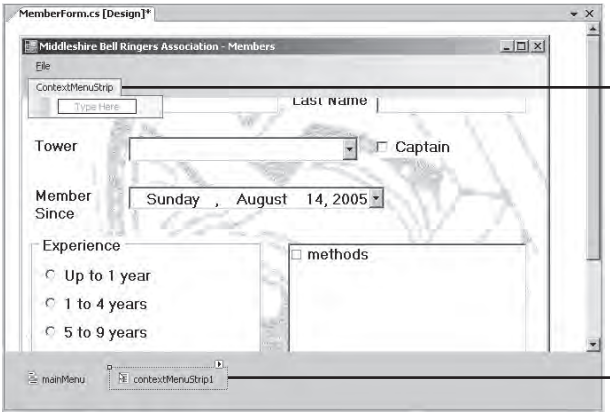
יצירת תפריטי קיצור

בתרגילים הבאים עליך ליצור שני תפריטי קיצור. תפריט הקיצור הראשון שייך לתיבות הטקסט firstName ו-lastName ומאפשר למשתמש לאפס את הפקדים. תפריט הקיצור השני שייך לטופס עצמו וכולל פקודות שנועדות לשמור את הנתונים שנמצאים בטופס ואחר כך לאפס אותו. כדי לעשות זאת, עליך ליצור עותק של פריט תפריט קיים, וגם ליצור פריט חדש.

צור את תפריטי הקיצור של firstName ו-lastName

1. בחלון תצוגת העיצוב המציג את **MemberForm**, גרור אל הטופס את הפקד **ContextMenuStrip** מתוך הקטגוריה Menus & Toolbars שבערכת הכלים.

   האובייקט **contextMenuStrip1** מסוג **ContextMenuStrip** יופיע בתחתית הטופס, ושורת תפריטים נוספת תופיע בחלקו העליון. שים לב שזהו מיקום זמני עבור שורת התפריטים החדשה; היא נמצאת שם רק כדי שתוכל לערוך אותה ולהוסיף לה פריטים. בסביבת ההרצה המיקום של שורת התפריטים החדשה יקבע באופן אוטומטי על פי מיקום סמן העכבר כאשר המשתמש לוחץ לחיצה ימנית.



2. בחר בפקד **contextMenuStrip1**. בתיבת הטקסט (**Name**) שבחלון Properties הקלד **textBoxMenu** והקש Enter.

3. לחץ על הכיתוב Type Here שתחת שורת התפריטים ContextMenuStrip אשר בראש הטופס (לא הפקד). הקלד **Clear Text** והקש Enter.

טיפ

אם שורת התפריטים ContextMenuStrip נעלמת מהטופס, לחץ על הפקד textBoxMenu שתחת הטופס כדי להחזיר אותה.

4. לחץ על תיבת הטקסט **firstName** שנמצאת ליד התווית First Name. בחלון Properties שנה את המאפיין **ContextMenuStrip** לערך **textBoxMenu**. המאפיין **ContextMenu** קובע את התפריט שיוצג, אם בכלל, כאשר המשתמש לוחץ לחיצה ימנית על הפקד.

5. לחץ על תיבת הטקסט **lastName**. בחלון Properties שנה את המאפיין **ContextMenuStrip** לערך **textBoxMenu**.

שים לב שניתן לשייך תפריט קיצור אחד לפקדים שונים.

6. לחץ על הפקד של תפריט הקיצור **textBoxMenu**. לחץ על הלחצן Events, ובתיבת הטקסט של האירוע **Opening** הקלד **textBoxContextMenuPopup** והקש Enter.

שיטת אירוע חדשה בשם **textBoxContextMenuPopup** תיווצר ותוצג בחלון Code and Text Editor. אירוע זה יוצף בכל פעם שיופיע תפריט הקיצור.

7. הוסף את המשפט הבא לשיטת האירוע **textBoxContextMenuPopup**:

```
this.Tag = ((ContextMenuStrip)sender).SourceControl;
```

הפרמטר **sender** של שיטת האירוע יהיה האובייקט **textBoxMenu**. אובייקט זה כולל מאפיין שימושי ביותר בשם **SourceControl**, אשר מפנה אל פקד שהמשתמש הפעיל דרכו את תפריט הקיצור. משפט זה מאחסן במאפיין **Tag** של הטופס את ההפניה אל תיבת הטקסט הנוכחית.

המאפיין **Tag** של הטופס הוא פריט לשימוש כללי, שיכול לשמש לאחסון של כל נתון שדרוש לתוכנית.

8. חזור לחלון תצוגת העיצוב. לחץ על הפקד **textBoxMenu** שבתחתית הטופס כדי להציג שוב את שורת התפריטים ContextMenuStrip.

9. בחלון Properties, לחץ Events, הקלד **textBoxClearClick** בתיבת הטקסט של האירוע **Click** והקש Enter.

שיטת אירוע חדשה בשם **textBoxClearClick** תיווצר, ותוצג בחלון Code and Text Editor.

10. הוסף את המשפטים הבאים לשיטת האירוע textBoxClearClick:

```
if (this.Tag.Equals(firstName))
{
   firstName.Clear();
   firstName.Focus();
}
else
{
   lastName.Clear();
   lastName.Focus();
}
```

המשפט **if** קובע איזו מבין תיבות הטקסט היא המקור לתפריט הקיצור, ומאפס אותה. השיטה **Focus** של פקד מציבה בו את הסמן. לחיצה ימנית על פקד אינה גורמת לסימונו ולהעברת הסמן אליו.

11. בתפריט Debug בחר Start Without Debugging.

הפרויקט יעבור הידור ויופעל.

12. כאשר מופיע הטופס, פתח את תפריט File ובחר New. הקלד שם בתיבות הטקסט First Name ו–Last Name.

13. לחץ לחיצה ימנית על תיבת הטקסט First Name.

בתפריט הקיצור תופיע פקודה אחת בלבד: Clear Text.

14. לחץ על הפקודה Clear Text.

תיבת הטקסט First Name תתרוקן.

15. הקלד שם בתיבת הטקסט First Name ועבור לתיבת הטקסט Last Name. לחץ לחיצה ימנית על תיבת הטקסט Last Name כדי לפתוח את תפריט הקיצור. לחץ על הפקודה Clear Text.

הפעם, תיבת הטקסט Last Name היא זו שתתרוקן בעוד שהשם הפרטי יישאר בתיבה שלו.

16. לחץ לחיצה ימנית במקום כלשהו בטופס.

מכיוון שרק לתיבות הטקסט First Name ו–Last Name יש תפריטי קיצור, הלחיצה לא תגרום להופעת תפריט קיצור.

17. סגור את הטופס.

לפני שתיצור את תפריט הקיצור השני, עליך להפעיל את הפריט Save Member, אשר נכון לעכשיו אינו עושה דבר. כאשר תבחר בו, הנתונים שעל הטופס יישמרו בקובץ. כעת נשמור אותם בקובץ טקסט רגיל בשם Members.txt שבתיקייה הנוכחית. בהמשך נכתוב קוד שיאפשר למשתמש לבחור בתיקיות ובשמות קבצים אחרים כרצונו.

כתוב את שיטת האירוע saveMemberClick

1. הצג את הטופס **MemberForm** בחלון תצוגת העיצוב. פתח את תפריט File ובחר Save Member.
2. בחלון Properties לחץ Events, בחר באירוע Click, הקלד **saveMemberClick** והקש Enter.
3. בחלון Code And Text Editor הוסף לראש הקובץ MemberForm.cs את משפט using הבא:

```
using System.IO;
```

4. חזור לשיטת האירוע **saveMemberClick** שנמצאת בסוף הקובץ MemberForm.cs. הוסף את המשפטים הבאים לגוף השיטה:

```
StreamWriter writer = new StreamWriter("Members.txt");
writer.WriteLine("First Name: " + firstName.Text);
writer.WriteLine("Last Name: " + lastName.Text);
writer.WriteLine("Tower: " + towerNames.Text);
writer.WriteLine("Captain: " + isCaptain.Checked);
writer.WriteLine("Member Since: " + memberSince.Text);
writer.WriteLine("Methods: ");
foreach(object methodChecked in methods.CheckedItems)
{
   writer.WriteLine(methodChecked.ToString());
}
writer.Close();

MessageBox.Show("Member details saved", "Saved");
```

בלוק זה של קוד יוצר אובייקט מסוג **StreamWriter** המשמש לכתיבת הטקסט בקובץ Member.txt. השימוש במחלקה **StreamWrite** דומה מאוד לאופן הדפסת טקסט על המסך באמצעות האובייקט **Console**. לשם כך עליך להשתמש בשיטה **WriteLine**.

החלק המורכב ביותר בקוד זה הוא המשפט **foreach**, המשמש לדפדוף בפקד **methods**. המחלקה **CheckedListBox** כוללת את המאפיין **CheckedItems** שמכיל את כל הפריטים שסומנו. המשפט foreach שולח כל אחד מהפריטים שסומנו אל האובייקט **StreamWriter**, אשר מוסיף אותם לקובץ.

טיפ

כאשר המשתמש מפעיל את הקובץ, עליו ללחוץ פעמיים על כל שיטת צלצול שברשימת הסימון: פעם אחת הוא עושה זאת כדי לבחור בשיטה, ופעם נוספת - כדי לסמן אותה. הפקד CheckedListBox כולל מאפיין נוסף בשם CheckOnClick. כאשר הערך של מאפיין זה הוא true, ניתן לסמן כל אחד מהסעיפים שברשימת הסימון באמצעות לחיצה אחת בלבד.

לאחר שכל הנתונים הועברו לטופס, האובייקט **StreamWriter** נסגר, ותיבת הודעה מחזירה משוב למשתמש. שים לב לרעיון טוב שכדאי לאמץ!

5. בתפריט Debug בחר Start Without Debugging, כדי לבנות ולהפעיל את התוכנית.
6. הוסף חבר ארגון חדש והזן עבורו נתונים. בתפריט File בחר Save Member. לאחר השהיה קצרה, תוצג ההודעה "Member details saved". לחץ OK וסגור את הטופס.
7. השתמש ב- Windows Explorer כדי לפתוח את התיקייה \My Documents\Microsoft Press\Visual CSharp Step by Step\Chapter 21\BellRingers\bin\Debug.

   בתיקייה תמצא קובץ בשם Members.txt.
8. לחץ לחיצה כפולה על הקובץ כדי להציג את תכולתו באמצעות Notepad. הקובץ אמור להכיל את פרטי החבר החדש.
9. סגור את Notepad וחזור אל Visual Studio 2005.

כעת ניתן להוסיף את תפריט הקיצור השני.

בשביל הגיוון, וגם כדי שתראה כמה קל ליצור תפריטי קיצור, בתרגיל הבא עליך לכתוב בעצמך את הקוד ליצירת התפריטים. המקום הטוב ביותר לכתוב בו את הקוד הוא הבנאי של הטופס.

יצירת תפריט ההקשר MemberForm

1. עבור אל תצוגת הקוד עבור **MemberForm**: בתפריט View בחר Code.
2. אתר את הבנאי עבור **MemberForm**. זוהי למעשה השיטה הראשונה במחלקה, ונקראת **MemberForm**.

   תפריט מורכב ממערך של פריטי תפריט. בדוגמה הבאה, תפריט הקיצור של הטופס כולל שני פריטים, Save Member ו-Clear.
3. בגוף הבנאי, מייד לאחר הקריאה לשיטה Reset, הוסף את המשפט הבא:

```
ToolStripMenuItem[] formMenuItemList = new ToolStripMenuItem[2];
```

משפט זה יוצר מערך גדול דיו לאחסון שני פריטי תפריט.

4. הפריט הראשון בתפריט הוא העתק של הפריט **saveMemberToolStripMenu** שיצרת קודם לכן. הוסף את הפריט למערך **formMenuItemList**:

```
formMenuItemList[0] = new ToolStripMenuItem("Save Member",
   null,new System.EventHandler(saveMemberClick));
```

הבנאי מגדיר את הכיתוב של פריט התפריט, את התמונה שיש להשתמש בה (במקרה זה **null**), ונציג המפנה אל שיטת האירוע שצריך להפעיל בעת אירוע **Click**. זו כמובן השיטה שיצרת בתרגיל הקודם.

אתה בוודאי תוהה מדוע לא ניתן רק להפנות אל **saveMemberToolStripMenu** שיצרת קודם במקום ליצור העתק שלו. יש לכך שתי סיבות: ראשית, ייתכן שתרצה לשנות את המאפיינים של מופע זה מבלי להשפיע על התפריט הראשי של הטופס;

שנית, אם לא תעתיק את פריט התפריט Save Member הוא ייעלם מהתפריט הראשי של הטופס ברגע שתיצור הפניה אליו מתפריט הקיצור.

5. ניתן ליצור כך גם את הפריט השני. בפרק 20 יצרת לחצן (שנקרא גם הוא Clear) שביצע את אותה הפעולה. את שיטת האירוע שיצרת עבור הלחצן תוכל למחזר עבור הפריט Clear שבתפריט. הוסף את המשפטים הבאים:

```
formMenuItemList[1] = new ToolStripMenuItem("Clear", null ,
   new System.EventHandler(clearClick));
```

6. הוסף את המשפטים הבאים:

```
ContextMenuStrip formMenu = new ContextMenuStrip();
formMenu.Items.AddRange(formMenuItemList);
```

הקוד יצור אובייקט **ContextMenuStrip** חדש ויצרף אליו את המערך שמכיל את פריטי התפריט Save Member ו-Clear.

7. שייך את תפריט הקיצור אל הטופס עצמו על ידי הוספת המשפט הבא:

```
this.ContextMenuStrip = formMenu;
this.ContextMenuStrip.Enabled = false;
```

תפריט הקיצור צריך להיות חסום מלכתחילה, כי המשתמש אינו יכול ממילא להזין נתונים של חבר כלשהו עד להקשה על New שבתפריט File.

8. אתר את השיטה **newClick**. הוסף בסופה את המשפט הבא אשר מאפשר (enables) את **formMenu**:

```
this.ContextMenuStrip.Enabled = true;
```

9. הדר והפעל את הפרויקט. צור חבר ארגון חדש והקלד עבורו נתונים. לחיצה ימנית בכל מקום בטופס, מלבד תיבות הטקסט First Name ו-Last Name, תגרום להצגת תפריט הקיצור. אם תבחר Clear, הטופס יציג שוב את ערכי ברירת המחדל שלו. אם תבחר Save, הנתונים שהזנת יישמרו בקובץ Members.txt.

10. סגור את הטופס.

שימוש בפקדי דו-שיח משותפים

כעת היישום של מצלצלי הפעמונים מאפשר למשתמש לשמור נתונים, אולם הנתונים נשמרים תמיד באותו קובץ, ועל כן מוחקים נתונים קודמים שאוחסנו בו. כמו כן, המשתמש עדיין אינו יכול להשתמש ביישום כדי להדפיס את הנתונים. הבה נפתור בעיות אלו.

יש מספר פעולות שכיחות שכדי לבצען המשתמש צריך לציין סוג מידע מסוים. לדוגמה, אם המשתמש רוצה להדפיס קובץ, בדרך כלל הוא מתבקש לציין באיזו מדפסת ברצונו להשתמש.

הוא גם יכול לציין או לערוך מאפיינים נוספים, כגון מספר העותקים. ייתכן ששמת לב שמספר רב של יישומים משתמשים בתיבת דו-שיח Print זהה. הסיבה לכך אינה חוסר יצירתיות מצד מפתחי היישומים, אלא שהדרישות מתיבת הדו-שיח הזו דומות מאוד עבור יישומים רבים. לכן, חברת Microsoft יצרה תקן עבור תיבת הדו-שיח (common dialog). זהו רכיב המצורף למערכת ההפעלה Windows אשר יכול לשמש אותך ביישומים שונים.

יש מספר רב של תיבות דו-שיח משותפות וביניהן תיבות דו-שיח לפתיחה ולשמירה של קבצים, בחירת צבעים וגופנים, בחירת תצוגת עמוד ותצוגה מקדימה לפני הדפסה. בעת בניית יישומים ב- Visual Studio 2005, ניתן להשתמש בכל אחת מתיבות הדו-שיח המשותפות האלו באמצעות פקדי הדו-שיח המשותפים.

שימוש בפקד SaveFileDialog

בתרגיל הבא עליך להשתמש בפקד **SaveFileDialog**. ביישום מצלצלי הפעמונים, כאשר המשתמש שומר את פרטי החבר בקובץ, היישום מבקש ממנו לציין את שם הקובץ ומיקומו באמצעות הפקד **SaveFileDialog**.

השתמש בפקד SaveFileDialog

1. הצג את הטופס **MemberForm** בחלון תצוגת העיצוב.
2. בערכת הכלים, הרחב את הקטגוריה Dialogs.
3. גרור את הפקד SaveFileDialog אל הטופס.
   הפקד בשם **saveFileDialog1** מופיע מתחת לטופס.
4. בחר בפקד **saveFileDialog1**. בחלון Properties, קבע את המאפיינים על פי הערכים המופיעים בטבלה הבאה.

מאפיין	ערך	תיאור
(Name)	saveFileDialog	שם הפקד.
AddExtension	True	כאשר הערך של מאפיין זה הוא **True**, תיבת הדו-שיח יכולה לצרף אל שם הקובץ הנבחר על ידי המשתמש את סיומת הקובץ שבמאפיין **DefaultExt**, אם המשתמש לא הוסיף סיומת בעצמו.
DefaultExt	txt	סיומת ברירת המחדל שיש להשתמש בה אם המשתמש לא הוסיף סיומת בעצמו.
FileName	השאר ריק	שם הקובץ הנוכחי. מחק את הערך כאשר אינך רוצה שהקובץ יסומן כברירת מחדל.
InitialDirectory	C:\	ספריית ברירת המחדל של תיבת הדו-שיח.

מאפיין	ערך	תיאור
OverwritePromp	True	כאשר הערך של מאפיין זה הוא **True**, המשתמש יוזהר אם ינסה לשמור בקובץ ששמו כבר קיים. כדי שנוהל זה יפעל, עליך לשנות גם את ערך המאפיין **ValidateNames** ל-**True**.
Title	Bell Ringers	המחרוזת המוצגת בכותרת של תיבת הדו-שיח.
ValidateNames	True	על פי מאפיין זה נקבע אם שמות הקבצים עוברים אישור. הוא משמש גם מספר מאפיינים אחרים, כגון OverwritePrompt. כאשר הערך של מאפיין זה הוא True, תיבת הדו-שיח תוודא ששמות הקובץ שהוקלדו על ידי המשתמש מורכבים מתווים תקניים בלבד.

.5 בחלון Code and Text Editor מוצג הקובץ MemberForm.cs. בסוף הקובץ מצא את השיטה **saveMemberClick**.

.6 הוסף את המשפטים הבאים לתחילת השיטה, סביב הקוד שיוצר את האובייקט StreamWriter ומשתמש בו:

```
DialogResult buttonClicked = saveFileDialog.ShowDialog();
if (buttonClicked.Equals(DialogResult.OK))
{
   // existing code
   StreamWriter writer = new StreamWriter("Members.txt");
   ...
   // existing code
   MessageBox.Show("Member details saved", "Saved");
}
```

.7 המשפט הראשון מציג את תיבת הדו-שיח Save File באמצעות השיטה **ShowDialog**. תיבת הדו-שיח Save File היא מודַלית (modal) ומשמעות הדבר היא שהמשתמש אינו יכול להשתמש בטפסים אחרים ביישום עד אשר יסגור את התיבה על ידי לחיצה על אחד מהלחצנים שלה. תיבות דו-שיח מודליות כוללות מאפיין **DialogResult** המציין איזה לחצן הופעל על ידי המשתמש (תיבת הדו-שיח Save כוללת לחצן Save ולחצן Cancel). השיטה **ShowDialog** מחזירה את ערך המאפיין **DialogResult**. אם המשתמש לחץ Save, ערך המאפיין **DialogResult** יהיה OK (ולא Save, כי אין כזה ערך במאפיין).

חשוב

הפקד **SaveFileDialog** מבקש מהמשתמש לבחור שם לקובץ שברצונו לשמור בו את הנתונים. הפקד אינו מבצע את השמירה עצמה - את הקוד המבצע את השמירה עליך לכתוב בעצמך.

.8 ערוך את המשפט אשר יוצר את האובייקט **StreamWriter**:

```
StreamWriter writer = new StreamWriter(saveFileDialog.FileName);
```

כעת השיטה תכתוב את הנתונים בקובץ שציין המשתמש, ולא בקובץ Member.txt.

9. בנה והפעל את היישום. צור חבר חדש. בתפריט File בחר Save Member. תיבת הדו-שיח Bell Ringer תיפתח ותתבקש לבחור שם עבור הקובץ שברצונך לשמור. אם לא תוסיף סיומת לקובץ, הקובץ הנשמר יקבל את הסיומת "txt." באופן אוטומטי. אם תבחר באחד מהקבצים הקיימים, תיבת הדו-שיח תזהיר אותך לפני שתיסגר שבכוונתך לכתוב על קובץ קיים. אשר זאת, אם זו כוונתך.
10. סגור את היישום.

באפשרותך להשתמש בטכניקה דומה גם לפתיחת קבצים. הוסף לטופס פקד **OpenFileDialog**, הפעל אותו באמצעות השיטה **ShowDialog**, ואם המשתמש לחץ Open, שלוף את המאפיין **FileName** כאשר השיטה מחזירה ערך. תוכל לפתוח את הקובץ, לקרוא את תכולתו ולהציבה בשדות שעל המסך.

פרטים נוספים על **OpenFileDialog** תוכל למצוא ב- MSDN Library for Visual Studio 2005.

שימוש במדפסת

הדפסה היא תכונה חשובה ביישומי Windows מקצועיים, כי היא מאפשרת לקבל פלט אשר נגיש לכל אחד. סביבת הפיתוח כוללת פקדים שמאפשרים לשלוח נתונים להדפסה בצורה מהירה ופשוטה. אחד מהפקדים האלה הוא Common Dialog, שגם מאפשר למשתמש לבחור במדפסת הרצויה לו. הפקד **PrintDocument**, למשל, מאפשר למתכנת לערוך את הנתונים אשר נשלחים למדפסת.

בתרגיל האחרון בפרק זה, עליך לכתוב פקודה Print שתשתמש בפקדים **PrintDialog** ו-**PrintDocument**.

השתמש בפקד PrintDialog

1. הצג את הטופס **MemberForm** בחלון תצוגת העיצוב.
2. הרחב את הקטגוריה Printing שבערכת הכלים.
3. גרור לטופס את הפקד **PrintDialog**.
   הפקד יופיע מתחת לטופס, בשם **printDialog1**.
4. לחץ על הפקד **printDialog1**. בחלון Properties שנה את המאפיין (Name) ל-printDialog.
5. בטופס **MemberForm**, פתח את תפריט File ובחר Print.
6. בחלון Properties לחץ Events. בחר באירוע **Click**, הקלד **printClick** והקש Enter.
7. בחלון Code and Text Editor, הוסף לשיטה printClick את המשפטים הבאים:

```
DialogResult buttonClicked = printDialog.ShowDialog();
if (buttonClicked.Equals(DialogResult.OK))
{
   // You will write this code shortly
}
```

פעולות אלו זהות לאלו שבוצעו עבור הפקד **SaveFileDialog**.

השתמש בפקד PrintDocument

1. חזור לחלון תצוגת העיצוב.
2. גרור אל הטופס **MemberForm** את הפקד **PrintDocument** מהקטגוריה Printing שבערכת הכלים.

   פקד נוסף יופיע מתחת לטופס, בשם **printDocument1**.
3. באמצעות החלון Properties שנה את שם הפקד ל-**printDocument**. מחק את תכולת המאפיין **DocumentName**.
4. לחץ על הפקד printDialog. בחלון Properties, שנה את המאפיין Document של הפקד לערך **printDocument**. אתה חייב לעשות זאת, כי הפקד **printDialog** משתמש בפקד **printDocument** כדי לקבל את הגדרות המדפסת.
5. חזור לחלון Code and Text Editor שמוצג בו MemberForm.cs, וחזור שוב לשיטה **printClick**.
6. החלף את ההערה שבגוף השיטה במשפט הבא:

```
printDocument.Print();
```

מטרת משפט זה להתחיל את תהליך ההדפסה במדפסת הנבחרת. אולם טרם סיימת, כי עליך לערוך את המידע שיודפס באמצעות האירוע **PrintPage** של הפקד **printDocument**.

7. בחלון תצוגת העיצוב לחץ על הפקד **printDocument**. בחלון Properties לחץ Events. בחר באירוע **PrintPage**, הקלד **printPage** והקש Enter.
8. בחלון code and Text Editor, הוסף לשיטה printPage את המשפטים הבאים:

```
StringBuilder data = new StringBuilder();
StringWriter writer = new StringWriter(data);
writer.WriteLine("First Name: " + firstName.Text);
writer.WriteLine("Last Name: " + lastName.Text);
writer.WriteLine("Tower: " + towerNames.Text);
writer.WriteLine("Captain: " + isCaptain.Checked);
writer.WriteLine("Member Since: " + memberSince.Text);
writer.WriteLine("Methods: ");
foreach (object methodChecked in methods.CheckedItems)
{
   writer.WriteLine(methodChecked.ToString());
}
writer.Close();
```

כעת אתה אמור לזהות חלק גדול מהקוד, כי הלוגיקה שלו דומה מאוד לזו ששימשה אותנו כאשר שמרנו את הנתונים בקובץ, בסיום הביצוע של בלוק הקוד, המשתנה data יכיל מידע מוכן להדפסה.

הערה

ב-NET Framework ובשפת #C, לא ניתן לשנות את סוג הנתון string. בכל פעם שאתה משנה את הערך במחרוזת, סביבת ההרצה יוצרת מחרוזת חדשה המכילה את הערך החדש, ולאחר מכן נפטרת מהמשתנה הישן. שינוי תכוף במחרוזות עלול להוות עול כבד על הזיכרון, ולפגוע ביעילות הקוד, כי כל שינוי מחייב יצירת מחרוזת חדשה והעברת המחרוזת הישנה אל מפנה האשפה לשחרור הזיכרון. המחלקה **StringBuilder** שבמרחב השמות **System.Text**, מעוצבת בצורה שנועדה להתגבר על הבעיה הזו. באפשרותך להוסיף ולהסיר תווים מאובייקטים מסוג StringBuilder בעזרת השיטות **Append**, **Insert** ו-**Remove**, מבלי ליצור אובייקט חדש בכל פעולה.

9. צרף את המשפטים הבאים בסוף השיטה **printPage**:

```
float leftMargin = e.MarginBounds.Left;
float topMargin = e.MarginBounds.Top;
float yPos = 0;
Font printFont = null;
printFont = new Font("Arial", 12);
yPos = topMargin + printFont.GetHeight(e.Graphics);
e.HasMorePages = false;
e.Graphics.DrawString(data.ToString(),printFont,Brushes.Black,
   leftMargin, yPos, new StringFormat());
```

בלוק קוד זה שולח את תוכן המשתנה **data** אל המדפסת. הנתונים יודפסו בגופן Arial, אך תוכל לבחור בכל גופן וגודל המותקנים במחשב שלך. רוחב השוליים של המסמך נקבע על פי ערך הפרמטר **PrintPageEventArgs** אשר מועבר לשיטה.

המידע נשלח אל המדפסת בעת הפעלת המשפט DrawString. תוכל לנסות להעביר כפרמטרים מספר ערכים שונים. הקוד המוצג לעיל יגרום להדפסת עמוד טקסט פשוט. אם תרצה, תוכל לנסות ולהוסיף כותרות וגרפיקות שונות לפלט.

10. בנה והפעל את היישום. צור חבר ארגון חדש.

11. בתפריט File בחר Print. בתיבת הדו-שיח Printer שתופיע, בחר במדפסת הרצויה ולחץ Print.

תיבת הודעה תודיע לך שהמידע עובד ונשלח להדפסה.

12. סגור את היישום.

☯ **אם ברצונך להמשיך לפרק הבא**

השאר את Visual Studio 2005 פעילה ועבור לפרק 22.

☯ **אם ברצונך לכבות כעת את Visual Studio 2005**

פתח את תפריט Menu ולחץ Exit. אם מופיעה תיבת דו-שיח Save, לחץ Yes.

פרק 21 – טבלה מסכמת

המשימה	צריך
ליצור תפריט עבור טופס.	להוסיף לטופס בקר **MenuStrip**.
להוסיף פריטים לתפריט.	ללחוץ על הבקר **MenuStrip** שבתחתית הטופס, ללחוץ על הכיתוב Type Here שבשורת התפריטים של הטופס ולהקליד את שם הפריט. כדי להוסיף פריטים יש להחליף את הכיתוב Type Here החדש. כדי להגדיר מקש גישה לפריט תפריט יש להוסיף את התו & לפני האות הרצויה בפריט.
ליצור קו הפרדה בתפריט.	ליצור פריט תפריט על ידי החלפת הכיתוב Type Here בסימן מינוס (-).
להגדיר מקש קיצור עבור פריט תפריט.	לבחור בפריט ולהגדיר את המאפיין **ShortcutKey** שבחלון Properties לפי שילוב המקשים הרצוי.
לאפשר (enable) ולחסום (disable) את פריטי התפריט.	בזמן העיצוב, יש לשנות את ערך המאפיין **Enabled** בחלון Properties ל-true או false. בסביבת ההרצה, יש להציב את הערך true או false במאפיין **Enabled** של פריט התפריט. לדוגמה, `printToolStripmenuItem.Enabled = true;`
לבצע פעולה (להגיב לאירוע) כאשר המשתמש בוחר באחד מפריטי התפריט.	לבחור בפריט הרצוי. בחלון Properties, ללחוץ Events. באירוע Click יש להקליד את שם שיטת האירוע. לבסוף יש לכתוב את הקוד בגוף השיטה שנוצרה.
ליצור תפריט קיצור.	להוסיף לטופס פקד **ContextMenuStrip**. אחר כך להוסיף פריטים לתפריט הקיצור כפי שמוספים פריטים לתפריט הראשי.
לשייך תפריט קיצור לטופס או לפקד.	לשנות את המאפיין **ContextMenuStrip** של פקד או טופס כדי שיפנה לתפריט הקיצור עצמו.
ליצור תפריט קיצור באופן דינמי.	ליצור ולמלא מערך של פריטי תפריט. אחר כך להשתמש במערך כדי ליצור תפריט קיצור. לבסוף יש לשנות את המאפיין **ContextMenuStrip** של הפקד או הטופס כדי שיפנה אל תפריט הקיצור.
לבקש מהמשתמש לבחור שם עבור הקובץ שהוא שומר.	להשתמש בפקד **SaveFileDialog**. יש להציג את תיבת הדו-שיח באמצעות השיטה **ShowDialog**. לאחר סגירת תיבת הדו-שיח, המאפיין **FileName** יכיל את שם הקובץ שהמשתמש בחר. ניתן להשתמש בשיטה **OpenFile** של תיבת הדו-שיח כדי לפנות אל הקובץ ולפתוח אותו.

המשימה	צריך
לשלוח מסמך למדפסת.	להשתמש בפקד **PrintDialog** כדי לבקש מהמשתמש לבחור במדפסת הרצויה. האירוע **PrintPage** של הפקד **PrintDocument** משמש לשליטה במשלוח המסמך למדפסת.

פרק 22

בדיקות תקינות

כאשר תסיים פרק זה, תוכל:

- לבדוק את המידע אשר המשתמש מזין, כדי לוודא שאינו מפר את חוקי היישום או חוקים עסקיים כלשהם.
- להשתמש במאפיין **CausesValidation** ולהבין את המגבלות החלות על האימות והתקינות (validation) של אירועים של טפסים ופקדים.
- לבצע בדיקות תקינות יעילות מבלי להפריע לפעולת התוכנית.
- להשתמש בפקד **ErrorProvider** כדי להציג הודעות שגיאה.
- להשתמש בפקד **StatusStrip** כדי להוסיף שורת מצב לתחתית הטופס.

בשני הפרקים האחרונים ראית כיצד ניתן ליצור יישום Windows Form שמפעיל פקדים שונים לקליטת נתונים, יצרת תפריטים המקלים על השימוש ביישום, וגם למדת ללכוד אירועים המופעלים על ידי תפריטים, טפסים ופקדים כדי שהיישומים שלך יוכלו להגיב לפעולות המשתמש. אין ספק שעיצוב מחושב של הטופס ושימוש נכון בפקדים יכול לתמוך בסינון הנתונים שהמשתמש מזין למערכת. עם זאת, לעיתים קרובות תצטרך לערוך בדיקות נוספות. בפרק זה תלמד כיצד לקבוע את תקינות הנתונים שהוקלדו על ידי המשתמש, כדי לוודא שהם עומדים בכללים העסקיים הכלולים בדרישות מהיישום.

בדיקת תקינות נתונים

אבטחת תקינות נתוני הקלט הינה לכאורה פשוטה למדי, אולם מימושה אינו פשוט תמיד, ובעיקר כאשר יש צורך בבדיקה משולבת של נתונים שהוזנו על ידי המשתמש באמצעות שני פקדים או יותר. ייתכן שהחוקים העסקיים הבסיסיים ישירים למדי, אולם לעיתים קרובות בדיקת תקינות הנתונים מבוצעת בדרך שהופכת את השימוש בטופס למסורבל, לעיתים אף מתסכל.

המאפיין CausesValidation

לטפסים ולפקדים של Windows יש מאפיין בוליאני בשם **CausesValidation** שתפקידו לקבוע אם הטופס או הפקד יפעילו את אירועי אי-תקינות הנתונים. כאשר הערך של המאפיין הוא **true** (ברירת המחדל), אז כאשר הפקד שהמאפיין שייך לו מקבל את המיקוד, כמו בעת לחיצה של המשתמש, הפקד הקודם שאיבד את המיקוד יעבור תהליכי בדיקת תקינות. במקרה כשל, המיקוד יחזור לפקד שנבדק. חשוב להבין שהמאפיין **CausesValidation** אינו משפיע על הפקד עצמו, אלא על שאר הפקדים שבטופס. אם הנושא עדיין אינו מובן לך, אל חשש - תבין הכל באמצעות הדוגמה שבהמשך.

אירועים לבדיקות תקינות נתונים

כדי לבדוק תקינות נתונים בפקד, ניתן להשתמש בשני אירועים: **Validating** (בודק תקינות) ו-**Validated** (תקין). האירוע Validating מתרחש כאשר המיקוד (focus) עוזב את אחד הפקדים ומנסה לעבור אל הפקד שערך המאפיין **CausesValidation** שלו הוא **true**. ניתן להשתמש באירוע **Validation** (בדיקת תקינות) כדי לבדוק את ערך הפקד שאיבד את המיקוד. אם הערך אינו עומד בציפיות, תוכל לשנות את המאפיין Cancel בפרמטר **CancelEventArgs** וכך למנוע את שינוי המיקוד. רצוי תמיד להודיע למשתמש על הסיבה לאי-התקינות של הנתונים.

האירוע **Validation** מתרחש בסיום האירוע **Validating**, כל עוד הוא לא נכשל, ולפני שהפקד מאבד את המיקוד. אינך יכול לבטל את האירוע הזה ולכן, למטרת בדיקת הקלט מהמשתמש, יעילותו פחותה מזו של האירוע **Validating**.

טיפ

האירוע **Validating** מוצף רק כאשר המיקוד עובר לפקד שערך המאפיין **CausesValidation** שלו הוא **true**. לכן, אין טעם ביצירת טופס שבו כמה מהפקדים הם בעלי המאפיין **false** ואחרים בעלי המאפיין **true**. במקרה כזה, בדיקת התקינות תהיה במקום שהמשתמש לוחץ על הפקד. עשה זאת רק אם יש לך סיבה טובה מאוד לכך, כי מצב כזה עשוי לבלבל את המשתמש ולגרום לקבלת נתונים שאינם עקביים.

דוגמה – רישום לקוחות

ניקח לדוגמה תרחיש פשוט לבניית יישום המבצע רישום לקוחות. חלק מהיישום מתעד את הפרטים החיוניים של הלקוח: תואר, שם ומין. נניח שבחרת ליצור טופס הדומה לזה שלהלן:

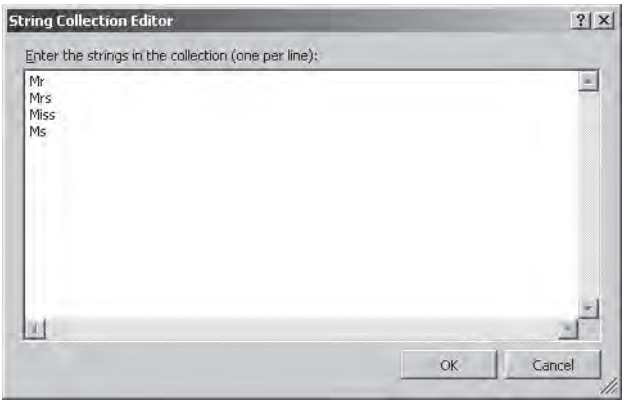
עליך לוודא שהנתונים שהמשתמש מזין הינם עקביים מבחינה זו שהתואר (Mr, Mrs, Miss או Ms) חייב להתאים למין הלקוח (זכר או נקבה) ולהפך.

ניסיון ראשון לבדיקת תקינות

בתרגיל הבא עליך לעיין בקוד של היישום לרישום הלקוחות ולהפעיל אותו, כדי לראות עד כמה קל לשבש בדיקות תקינות.

בדוק את התוכנית

1. פתח את הפרויקט **CustomerDetails** שבתיקייה My Documents\Microsoft Press\ Visual CSharp Step By Step\ Chapter 22\CustomerDetails.
2. ב- Solution Explorer, לחץ לחיצה כפולה על CustomerForm.cs כדי להציג את הטופס Customer Details בתצוגת העיצוב.
3. לחץ על התיבה המשולבת Title שבטופס, ובחלון Properties בחר במאפיין **Items** שלה. הוא אמור להופיע כאוסף (Collection). לחץ על לחצן ההמשך (Ellipses button) כדי להציג את המחרוזות שבאוסף.



מאוסף זה ניתן להסיק שתיבת הרשימה כוללת ארבעה תארים: Mr, Mrs, Miss או Ms. לחץ Cancel כדי לסגור את החלון String Collection Editor.

4. בדוק את תיבת הקבוצה Gender ואת לחצני האפשרויות. תיבת הקבוצה מכילה שני לחצני אפשרויות, Male ו-Female.

   היישום אוכף את החוק שעל פיו התואר והמין חייבים להתאים זה לזה. אם התואר הוא Mr, המין חייב להיות זכר; ואם התואר הוא Mrs, Miss או Ms, המין חייב להיות נקבה.

5. בתפריט View בחר Code כדי לעבור לחלון Code and Text Editor שמוצג בו הקובץ CustomerForm.cs. בחן את השיטה **checkTitleAndGender** שנמצאת מייד לאחר הבנאי **CustomerForm**:

```
if (title.Text == “Mr”)
{
   // Check that gender is Male
   if (!male.Checked)
   {
```

```
        MessageBox.Show("If the title is Mr the gender must be
male", "Error", MessageBoxButtons.OK, MessageBoxIcon.Error);
        return false;
      }
   }
   else
   {
      // Check that the gender is Female
      if (!female.Checked)
      {
        MessageBox.Show("If the title is Mrs, Miss, or Ms
the gender must be female", "Error", MessageBoxButtons.OK,
MessageBoxIcon.Error);
        return false;
      }
   }
   // Title and gender match
   return true;
}
```

שיטה זו מבצעת בדיקה צולבת פשוטה, בין הערך שבתיבה המשולבת Title לבין לחצני האפשרויות שבתיבת הקבוצה Gender. אם התואר הוא "Mr", על פי החוק הכלל, מין הלקוח חייב להיות זכר, ולכן השיטה בודקת שלחצן האפשרויות Male אכן מסומן. אם הוא אינו מסומן, השיטה מציגה הודעת שגיאה ומחזירה ערך **false**. בנוסף, אם המשתמש בוחר באחד התארים האחרים (Mrs, Miss או Ms), אז על פי הכלל שקבענו, מין הלקוחה חייב להיות נקבה, והשיטה בודקת שלחצן האפשרויות Female מסומן. גם במקרה זה, אם הלחצן אינו מסומן תופיע הודעת שגיאה אחרת, והשיטה תחזיר ערך **false**. אם התואר והמין תואמים זה לזה, השיטה תחזיר ערך **true**.

6. התבונן בשיטות **titleValidating** ו-**genderValidating** שבסוף הקובץ. הן מימושים של מטפל האירוע **Validating** עבור התיבה המשולבת Title ותיבת הקבוצה Gender. שתי השיטות קוראות לשיטה **checkTitleAndGender**. הן מציבות ערך true במאפיין **Cancel** של הפרמטר **CancelEventArgs** אם השיטה **checkTitleAndGender** מחזירה ערך **false**.

```
private void titleValidating(object sender, CancelEventArgs e)
{
   if (!checkTitleAndGender())
   {
      e.Cancel = true;
   }
}
private void genderValidating(object sender, CancelEventArgs e)
{
   if (!checkTitleAndGender())
   {
      e.Cancel = true;
   }
}
```

7. עיין בשיטות הנותרות, **exitClick** ו-**saveClick**. בטופס תראה את שורת התפריטים שבה נמצא התפריט File שמכיל את הפריטים Save ו-Exit. השיטות נקראות כאשר המשתמש בוחר באחד משני הפריטים שבתפריט. השיטה **closeClick** גורמת לסגירת הטופס ויציאה מהיישום. השיטה **saveClick** גורמת להצגת ההודעה "Customer Saved" (מכיוון שאין זו גרסה מלאה של היישום, השיטה אינה שומרת את המידע).

כעת הפעל את היישום, וראה מה קורה כאשר נערכת בדיקת תקינות לקלט של המשתמש.

הפעלת היישום

1. בתפריט Debug בחר Start Without Debugging כדי להפעיל את היישום.

   כעת יוצג הטופס Customer Details. שים לב שמין ברירת המחדל הוא Male.

2. בתיבה המשולבת Title בחר Mrs והקש tab, כדי לעבור לתיבת הטקסט Name הראשונה.

   כעת תופעל השיטה **checkTitleAndGender** ותציג הודעת שגיאה, מכיוון שהתואר והמין אינם מתאימים.

3. לחץ OK בתיבת הדו-שיח Error כדי לסגור אותה, ונסה ללחוץ על לחצן האפשרויות Female שבתיבת הקבוצה.

   לא תצליח לעשות זאת, מכיוון שהמאפיין **CausesValidation** של תיבת הקבוצה גורם לביצוע חוזר של האירוע **Validating**, ואז הודעת השגיאה תופיע שוב.

4. לחץ OK בתיבה Error. שנה את התואר חזרה ל-Mr וסמן את לחצן האפשרויות Female.

   זכור שהאירוע **Validating** מופעל לפני שהמיקוד עובר ללחצן האפשרויות Female. כאשר מתבצעת הקריאה לשיטה **checkTitleAndGender**, התואר (Mr) והמין (Male) מתאימים, ולכן תוכל לסמן את לחצן האפשרויות Female.

5. לאחר ששינית את המין לנקבה, נסה לשמור את פרטי הלקוח באמצעות הפריט Save שבתפריט File. השמירה תעבור ללא תקלות, וההודעה Customer Saved תוצג. מבולבל? לחץ OK.

6. נסה לתקן את השגיאה (שלא דווחה בכלל) על ידי שינוי התואר ל-Mrs.

   האירוע **Validating** יופעל (הפעם, עבור תיבת הקבוצה Gender), יזהה שהמין והתואר אינן מתאימים זה לזה, ויציג הודעת שגיאה. כדי להימנע משגיאה עליך לשנות את המין חזרה ל-Male (כפי שהיה בהתחלה).

7. צא מהיישום.

אסטרטגיה זו לביצוע תקינות נתונים נכשלה, מכיוון שהאירוע **Validating** מופעל רק במעבר בין שני פקדים באותו טופס, ולא במעבר לפקד שנמצא בסרגל הכלים או בשורת התפריטים, לדוגמה. לעיתים קרובות מתכנתים צריכים להשקיע זמן רב כדי שבדיקת התקינות תפעל כשורה. כעת נלמד להשתמש בכלים שבידנו כדי לקבוע תקפות נתונים בדרך הנכונה.

טיפ

עליך להשתמש באירוע **Validating** רק לשם בדיקת תקינות בפקדים יחידים. אל תשתמש בו כדי לבדוק תוכן של פקד אחד ביחס לתוכן של פקד אחר.

הדרך להבטיח פעולה תקינה

ביישום של רישום לקוחות יש קושי, בכך שבדיקת התקינות מבוצעת בנקודת זמן לא נכונה. היא מבוצעת באופן בלתי עקבי ומתערבת יותר מדי בפעולת התוכנית. ההיגיון שלעצמו הוא טוב, אך עלינו למצוא גישה אחרת, טובה יותר, לביצוע בדיקת תקינות.

פתרון טוב יותר יהיה לבדוק את הקלט בעת שמירת הנתונים. כך ניתן לדעת שהמשתמש סיים להזין את כל הנתונים ושהם אמורים להיות, מבחינתו, עקביים. אם מתגלית בעיה כלשהי, הודעת שגיאה תעדכן את המשתמש ותמנע את שמירת הנתונים עד אשר יתוקנו. בתרגיל הבא, עליך לשנות את היישום לרישום הלקוחות כדי שבדיקת התקינות תבוצע בעת שמירת פרטי הלקוח.

שינוי עיתוי בדיקת תקינות הנתונים

1. חזור לטופס **CustomerForm** שבתצוגת העיצוב. לחץ על התיבה המשולבת Title.
2. בחלון Properties לחץ Events. אתר את האירוע **Validating** ומחק את השיטה **titleValidating**.

   פעולה זו גורמת להסרת התיבה המשולבת Title מהאירוע **Validating**.
3. בטופס **CustomerForm** לחץ על תיבת הקבוצה Gender.
4. בחלון Properties לחץ Events. אתר את האירוע **Validating** ומחק את השיטה **genderValidating**.

חשוב

ביטול השיוך לאירועים בצורה כזו מנתקת את שיטת האירוע מהאירוע עצמו, אולם אינה גורמת למחיקתה. עם זאת, אם אינך זקוק לשיטה, תוכל להסיר אותה באופן ידני בחלון Code and Text Editor.

5. הצג את הקובץ CustomerForm.cs בחלון Code and Text Editor. מחק את השיטות **titleValidating** ו-**genderValidating**. עליך לקרוא לשיטה **checkTitleAndGender** כאשר המשתמש לוחץ Save.
6. אתר את השיטה **saveClick**. זהו המיקום הנכון לקוד שבאמצעותו תבצע בדיקת תקינות לנתונים. ערוך את השיטה באופן הבא:

```
private void saveClick(object sender, EventArgs e)
{
   if (checkTitleAndGender())
   {
      // Save the current customer's details
      MessageBox.Show("Customer saved", "Saved");
   }
   else
   {
      MessageBox.Show("Customer title and gender are
        inconsistent" +" - please correct and save again",
        "Error", MessageBoxButtons.OK, MessageBoxIcon.Error);
   }
}
```

7. המשפט **if** קורא לשיטה **checkTitleAndGender**. אם השיטה מחזירה ערך **true**, אז השיטה **saveCustomerForm** תציג את תיבת ההודעה "Customer Saved". אם השיטה **checkTitleAndGender** מחזירה ערך false, תופיע הודעת שגיאה ופרטי הלקוח לא יישמרו.

בדוק את היישום בשנית

1. בנה והפעל את היישום. בטופס Customer Details שנה את התואר שבתיבה המשולבת ל-Mrs, ולחץ על תיבת הטקסט Name הראשונה. פעולה זו לא תגרום לשגיאה, מכיוון שהתיבה המשולבת Title כבר אינה מפעילה את האירוע **Validating**.
2. ודא שלחצן האפשרויות Male מסומן, ובתפריט File בחר Save. בשלב זה תבוצע קריאה לשיטה **checkTitleAndGender** וחוסר ההתאמה מדווח למשתמש. לחץ OK.
3. שים לב להופעת תיבת הודעה נוספת, שמגיעה מהשיטה **saveClick**. לחץ OK גם הפעם.
4. סמן את לחצן האפשרויות Female ועל הלחצן Save Customer שבסרגל הכלים.
   הפעם לא תופיע הודעת שגיאה, אלא הודעה על שמירת פרטי הלקוח.
5. לחץ OK וצא מהיישום.

שימוש בפקד ErrorProvider

דחיית בדיקת התקינות הינה פעולה חיובית המקלה על השימוש בטופס. אבל מה קורה אם מתגלות מספר שגיאות בעת הבדיקה המבוצעת בעת שמירת המסמך? אם הנך משתמש בתיבות הודעה כדי לדווח למשתמש על שגיאות, היישום עשוי להציג מספר הודעות ברצף במקרה של שגיאות מרובות. כמו כן, המשתמש צריך לזכור את כל השגיאות כדי שיוכל לתקנן. כאשר מספר השגיאות גדול משתיים או שלוש, הדבר עשוי להקשות עליו. למרבה המזל, שימוש בפקד **ErrorProvider** יכול לסייע בפתרון הבעיה.

הוסף פקד ErrorProvider

1. חזור לטופס **CustomerForm** שבחלון תצוגת העיצוב.
2. בערכת הכלים הרחב את הקטגוריה Components. לחץ על הפקד **ErrorProvider** וגרור אותו אל הטופס.
   הפקד יופיע תחת הטופס המוצג.

3. לחץ על הפקד **errorProvider1** ועבור לחלון Properties. שנה את המאפיין (**Name**) ל-**errorProvider** וודא שהמאפיין **BlinkStyle** אכן מקובע על **BlinkIfDifferentError**.

כאשר אחד מהפקדים מדווח על שגיאה, סמל שגיאה שנקבע על פי המאפיין Icon, יופיע לצד הפקד שהניב את השגיאה, יהבהב לזמן קצר ויישאר שם. ניתן לשלוט בקצב ההבהוב באמצעות המאפיין **BlinkRate** - ערך ברירת המחדל הוא 250 אלפיות השנייה (4 הבהובים בשנייה). בעת דיווח על שגיאה נוספת, הסמל יהבהב רק אם השגיאה שונה מהשגיאה הקודמת. אם אתה רוצה שהסמל יהבהב בכל הופעת שגיאה, שנה את המאפיין **BlinkStyle** ל-**AlwaysBlink**.

בטופס **CustomerForm**, בחר בתיבה המשולבת Title. אם תביט בחלון Properties תבחין במאפיין חדש **Error on errorProvider**. מאפיין זה יופיע רק לאחר הוספת בקר **ErrorProvider** לטופס. אם תקליד הודעה במאפיין (כגון "Testing"), סמל שגיאה יופיע לצד התיבה המשולבת Title. אם תזיז את סמן העכבר מעל לסמל השגיאה, תופיע מסגרת קטנה המציגה את הודעת השגיאה.

אם תשאיר את המאפיין במצב זה, סמל השגיאה יישאר על הטופס באופן קבוע. כדי שהסמל יופיע רק במקרה של שגיאה ובשילוב עם הודעת שגיאה בעלת משמעות, עליך למחוק את הטקסט שבמאפיין **Error on errorProvider**. כעת עליך לכתוב קוד אשר יפעיל את הפקד **errorProvider** באופן דינמי.

4. הצג את הקוד עבור **CustomerForm** בחלון Code and Text Editor. אתר את השיטה **checkTitleAndGender** והחלף את המשפט הקורא לשיטה **MessageBox.Show** בקריאה לשיטה **errorProvider.SetError** באופן הבא:

```
// Cross check the gender and the title to make sure they
correspond private bool checkTitleAndGender()
{
   if (title.Text == “Mr”)
   {
      // Check that gender is Male
      if (!male.Checked)
      {
         errorProvider.SetError(gender, “If the title is Mr “ +
            “the gender must be male”);
         errorProvider.SetError(title, “If the gender is “ +
            “female the title must be Mrs, Miss, or Ms”);
         return false;
      }
   }
   else
   {
      // Check that the gender is Female
      if (!female.Checked)
      {
         errorProvider.SetError(gender, “If the title is Mrs, “
             + “Miss, or Ms the gender must be female”);
         errorProvider.SetError(title, “If the gender is male “
             + “the title must be Mr”);
         return false;
      }
   }
   // Title and gender match - clear any errors
   errorProvider.SetError(gender, “”);
   errorProvider.SetError(title, “”);
   return true;
}
```

השיטה **SetError** של הפקד **ErrorProvider** קובעת את הפקד שיש לסמן באמצעות סמל השגיאה, ואת ההודעה שיש להציג במסגרת ToolTip. אם תעביר לפרמטר השני מחרוזת ריקה, כפי שמבוצע בקוד שבסוף השיטה, סמל השגיאה יוסר מהטופס.

בדוק את הפקד ErrorProvider

1. בנה והפעל את היישום.
2. בתיבה המשולבת Title בחר Mrs וודא שלחצן האפשרויות Male מסומן.
3. בתפריט File בחר Save. על המסך תוצג הודעת שגיאה המודיעה על כישלון תהליך השמירה, וסמלי שגיאה יופיעו ליד הפקדים הקשורים לשגיאה. לחץ OK. אם תציב את סמן העכבר מעל אחד מהפקדים, תראה את הודעת השגיאה, כפי שמוצג במסך הבא:

לעומת היישום המקורי, דרך זו לקביעת התקפות אמינה ועקבית יותר, ובאותו זמן היא גם מפריעה פחות לפעולה הסדירה של היישום.

4. סמן את לחצן האפשרויות Female. בתפריט File בחר Save.

מכיוון שכעת הנתונים עקביים, סמלי השגיאה ייעלמו, ותוצג הודעה המדווחת על שמירה מוצלחת.

5. צא מהיישום.

הוספת שורת מצב

למרות שהצלחנו להיפטר מחלק מתיבות ההודעה, חלק מהן עדיין נותרו - תיבת הודעה מוצגת כאשר פרטי הלקוח נשמרים בהצלחה, או במקרה של שגיאה. המשוב שבהודעות אלו צריך להגיע למשתמש, אולם מצד שני זה מציק ללחוץ OK כל הזמן כדי לאשר אותן. דרך טובה יותר ליידע את המשתמש על מצבים אלה היא להציג את ההודעות בשורת מצב בתחתית הטופס. המשתמש יוכל לקרוא אותן, אולם הוא לא יצטרך ללחוץ על שום דבר כדי להעלימן.

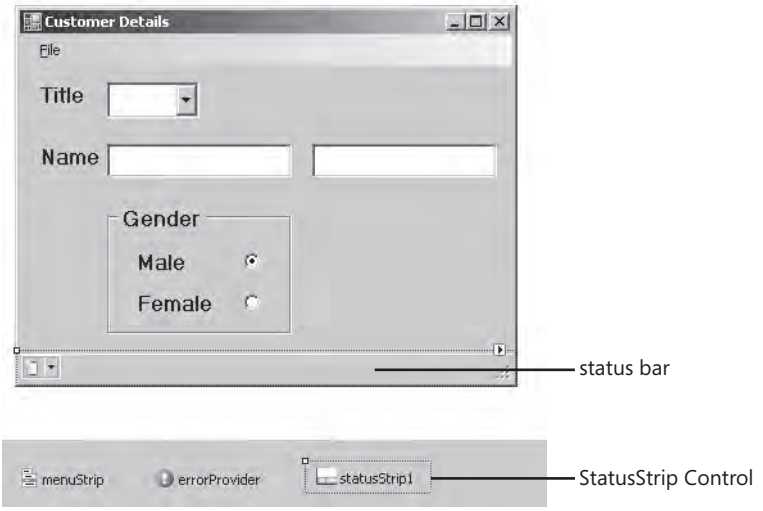
בתרגיל האחרון לפרק זה, עליך לממש שורת מצב בטופס **CustomerForm** באמצעות הפקד **StatusStrip**.

הוספת פקד StatusStrip

1. חזור לטופס **CustomerForm** שבחלון תצוגת העיצוב.

2. בערכת הכלים, הרחב את הקטגוריה Menus & Toolbars. גרור את הפקד **StatusStrip** מסרגל הכלים אל הטופס.

הפקד יוצג תחת הטופס, ושורת מצב תיווסף לחלק התחתון של הטופס.



3. לחץ על הפקד **statusStrip1**, ועבור לחלון Properties. שנה את המאפיין (**Name**) ל-**statusStrip**.

4. בשורת המצב שבטופס, לחץ על החץ שבמסך הבא, ובתפריט הנפתח בחר StatusLabel.

פעולה זו תגרום להוספת פקד **ToolStripStatusLabel** לשורת המצב. זו גרסה של הפקד **Label** המיועדת לשימוש בשילוב עם סרגלי כלים ושורות מצב.

5. לחץ על הפקד **toolStripStatusLabel1** שבשורת המצב. שנה את שמו ל-**statusMessages** באמצעות חלון Properties. מחק את תוכן המאפיין **Text**.
הפקד יקטן, אך הוא לא ייעלם משורת המצב.

6. הצג את **CustomerForm** בחלון Code and Text Editor. אתר את השיטה **saveClick**. שיטה זו מכילה את המשפטים אשר מחוללים את **תיבות ההודעה** שעליך להמיר לכיתוב בשורת המצב.

7. ערוך את המשפט **MessageBox** הראשון (הוא נמצא במשפט **if**), באופן שיציג את ההודעה בשורת המצב:

```
statusMessages.ForeColor = Color.Black;
statusMessages.Text = “Customer Saved”;
```

בסעיף הבא תבין מדוע עליך לשנות את צבע הטקסט לשחור.

8. שנה את המשפט **MessageBox** השני (הוא נמצא במשפט **else**), כדי שיציג את הודעת השגיאה בשורת המצב:

```
statusMessages.ForeColor = Color.Red;
statusMessages.Text = “Customer title and gender are
  inconsistent. Changes not saved”;
```

הודעת השגיאה מוצגת באדום כדי להבליט אותה. בעת הצגה של הודעה רגילה (כגון "Customer Saved", עליך לשנות את הצבע חזרה לשחור.

בדיקת שורת המצב

1. בנה והפעל את היישום. אם אתה זקוק לגרסה מלאה של היישום המוכן, תוכל למצוא אותה בתיקייה My Documents\Microsoft Press\Visual CSharp Step By Step\ Chapter 22\ CustomerDetails Complete.

2. בתיבה המשולבת Title בחר Mrs, וודא שלחצן האפשרויות Male מסומן.

3. בתפריט File בחר Save.

הודעת שגיאה אדומה, המודיעה על כישלון השמירה, תופיע בשורת המצב. סמלי שגיאה יופיעו לצד הפקדים האחראים לשגיאה כמקודם.

4. סמן את לחצן האפשרויות Female ובתפריט File בחר Save.

בשורת המצב תופיע ההודעה Customer Saved (בשחור) וסמלי השגיאה ייעלמו.

5. סגור את היישום.

☯ **אם ברצונך להמשיך לפרק הבא**

השאר את Visual Studio 2005 פעילה ועבור לפרק 23.

☯ **אם ברצונך לכבות כעת את Visual Studio 2005**

פתח את תפריט Menu ולחץ Exit. אם מוצגת תיבת דו-שיח Save, לחץ Yes.

פרק 22 – טבלה מסכמת

המשימה	צריך

לבצע בדיקת תקינות נתונים של פקד יחיד.	להשתמש בשיטת האירוע **Validating**. לדוגמה:
	`private void titleValidating(`
	`object sender,CancelEventArgs e)`
	`{`
	`  if (!checkTitleAndGender()  )`
	`  {`
	`  e.Cancel = true;`
	`  }`
	`}`

לאפשר ביצוע של אירוע **Validating**.	לשנות את ערך המאפיין **CausesValidation** של כל הפקדים בטופס ל-**true**.
לבצע בדיקת תקינות לתוכן של מספר פקדים או טופס שלם.	להשתמש בבדיקות ברמת הטופס. עליך ליצור שיטה לבדיקת תקינות לכל הנתונים שבטופס. קרא לה לאחר שהמשתמש סיים להזין את כל הנתונים, כאשר הוא בוחר לשמור אותם, למשל.
להודיע למשתמש איזה ערכים אחראיים לשגיאה, באופן שלא יפריע לפעולת התוכנית.	להשתמש בפקד **ErrorProvider**. קרא לשיטה **SetError** של הפקד **ErrorProvider** כדי שתציג סמל שגיאה והודעת שגיאה שניתן להציגה במסגרת ToolTip כאשר המשתמש מציב את סמן העכבר מעל סמל השגיאה.
להציג הודעות בשורת מצב בתחתית הטופס.	להוסיף פקד **StatusStrip** לטופס ולהוסיף פקד **ToolStripStatusLabel** לשורת המצב. את ההודעה שצריך להציג בשורת המצב בזמן ריצה יש להציב במאפיין **Text** של הפקד **ToolStripStatusLabel**.

חלק ה'

ניהול נתונים

בחלק זה:

פרק 23

עבודה עם בסיסי נתונים

כאשר תסיים פרק זה, תוכל:

- להתחבר אל בסיס נתונים באמצעות האשף Data Source Configuration.
- לאחזר נתונים מבסיס נתונים ולעיין באמצעות הממשק הגרפי של Visual Studio 2005.
- לאחזר נתונים מבסיס נתונים מסוג SQL Server באמצעות ADO.NET.

בחלק ד' של ספר זה, למדת כיצד להשתמש בשפת #C כדי לבנות ממשק משתמש ולהציג מידע. בחלק זה, תלמד לנהל נתונים באמצעות כלי גישה לנתונים שזמינים עבורך ב- Visual Studio 2005 וב- NET Framework. הפרקים הבאים עוסקים ב-ADO.NET, משפחה של אובייקטים המעוצבים במיוחד כדי להקל על כתיבת יישומים שמתבססים על בסיסי נתונים. אם בנית בעבר יישומים באמצעות Microsoft Visual Basic 6 או Microsoft Access, תראה ש-ADO.NET היא למעשה גרסה מעודכנת של ADO (ActiveX Data Objects), שמעוצבת במיוחד עבור סביבת ההרצה NET Framework.

חשוב

כדי לבצע את התרגילים שבפרק זה, עליך להתקין את היישום Microsoft SQL Server 2005 Express Edition אשר כלול ב- Visual Studio 2005. תוכל להוריד חינם מהאתר **http://www.Microsoft.com/sql/2005/default.asp** של Microsoft.

הערה

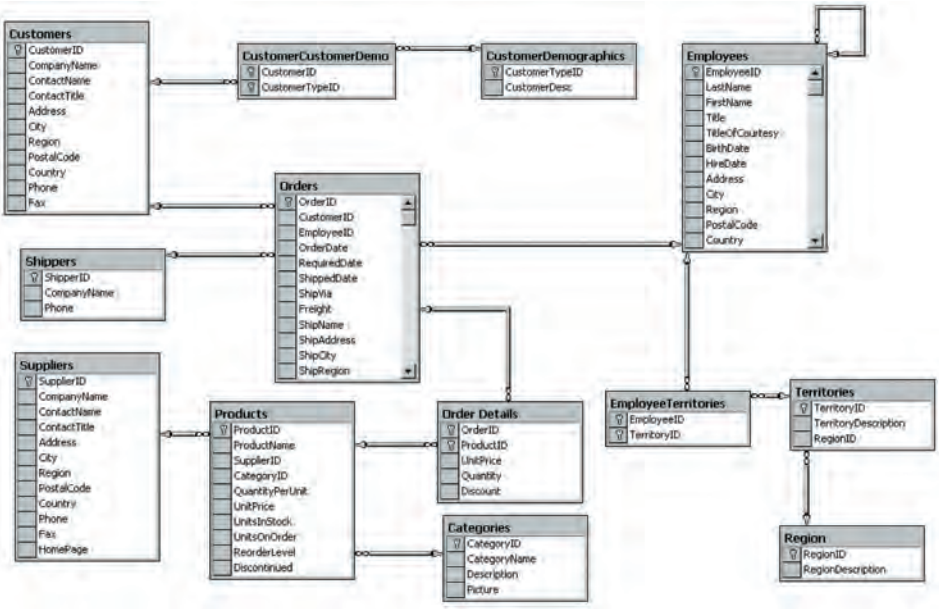
מידע והרחבה בנושא שרת SQL תוכל למצוא בספר SQL Server 2005 שבהוצאת הוד-עמי.

עבודה עם בסיסי הנתונים של ADO.NET

לאחר הצלחת NET Framework, החליטו בחברת Microsoft לעדכן את המודל שלהם לגישה אל בסיסי נתונים מסוג ActiveX Data Objects, וליצור את ADO.NET. מודל הגישה החדש כולל מספר שיפורים לעומת קודמיו, אשר תורמים לביצועים ולגמישות. אם כבר עבדת עם ADO, תיווכח שמודל האובייקטים של ADO.NET שונה במקצת. הסוג RecordSet אינו קיים יותר, כי Microsoft ייצרה את המחלקות TableAdapter ו-DataSet התומכות בגישה לנתונים ועבודה עם נתונים בתצורה מנותקת (disconnected). כך היא איפשרה לגשת לבסיס הנתונים מבלי להיות מחוברים כל הזמן לרשת, ועל ידי כך - לחסוך במשאבים. על ידי מנגנון שיתוף החיבורים (connection pooling) של ADO.NET, ניתן לשרת יישומים שונים, ולהפחית את הצורך בהתחברויות והתנתקויות חוזרות ונשנות לבסיס הנתונים, וכך לחסוך בפעולות גוזלות זמן. המודל של ADO.NET נועד להיות פשוט לשימוש, ואמנם, Visual Studio 2005 כוללת מספר אשפים וכלים אשר יכולים לסייע בבניית קוד הגישה לנתונים.

בסיס הנתונים של Northwind

"סוחרי Northwind" היא חברת דמה, אשר מוכרת דברי מאכל הנושאים שמות אקזוטיים. בסיס הנתונים של החברה מכיל מספר טבלאות שיש בהן מידע על המוצרים שמשווקים על ידי החברה, לקוחות החברה, הזמנות מלקוחות, ספקים שהחברה רוכשת מהם את הסחורות, מובילים שבאמצעותם הסחורה מופצת אל הלקוחות, והעובדים המועסקים בחברה. בתרשים 23-1 מוצגות כל הטבלאות שבבסיס הנתונים של סוחרי Northwind והקשרים שביניהן. בפרק זה נעסוק בטבלאות Orders ו-Products.



תרשים 23-1: טבלאות בסיס הנתונים של סוחרי Northwind

יצירת בסיס הנתונים

בתחילה עליך ליצור את בסיס הנתונים של סוחרי Northwind.

צור את בסיס הנתונים

1. בתפריט Start של Windows פתח את תפריט All Programs, Accessories ובחר Command Prompt כדי לפתוח חלון פקודות. בחלון הפקודות היכנס לתיקייה My Documents\Microsoft Press\Visual CSharp Step by Step\Chapter 23.

2. בחלון הפקודות הקלד את הפקודה הבאה:

```
sqlcmd -S YourServer\SQLExpress -E -i instnwnd.sql
```

במקום **YourServer** כתוב את שם המחשב שלך.

טיפ

אם אינך יודע את שם המחשב שלך, השתמש בפקודה **hostname** בחלון הפקודות, לפני הפעלת הפקודה **sqlcmd**.

פקודה זו מפעילה את תוכנית השירות sqlcmd כדי להתחבר למופע המקומי של SQL Server 2005 Express, ולהפעיל את התסריט instnwnd.sql. תסריט (script) זה מכיל את הפקודות SQL שבעזרתן ניצור את בסיס הנתונים של סוחרי Northwind ואת הטבלאות שבו, ונשמור בהן נתונים.

טיפ

לפני שתנסה ליצור את בסיס הנתונים של סוחרי Northwind עליך לוודא שהיישום SQL Server 2005 Express אכן פועל. כדי לבדוק אם היישום פעיל, או כדי להפעילו במידת הצורך, עליך להשתמש ביישום SQL Configuration Manager שבתיקייה Configuration Tools של קבוצת היישומים SQL Server 2005 CTP של Microsoft.

אם מכל סיבה שהיא הפקודה נכשלה, פתח את SQL Configuration Manager, יופיע חלון התחברות לבסיס הנתונים וודא כי בשדה (תיבה נגללת) Server Name: מוצג: **computer name**\SQLEXPRESS (שבו **computer name** הינו שם המחשב). לחץ על File, Open, File – בחר את הקובץ instnwnd.sql מהתיקייה My Documents\Microsoft Press\Visual CSharp Step by Step\Chapter 23, לחץ על Connect, לפניך יופיע תסריט המכיל משפטי פקודות אשר ייצרו את בסיס הנתונים, לחץ Query, Execute (או לחילופין F5), כדי להפעיל את התסריט.

3. לאחר סיום ביצוע הוראות התסריט, סגור את חלון הפקודות.

גישה אל בסיס הנתונים

בתרגילים הבאים תצטרך לכתוב תוכנית שמתחברת לבסיס הנתונים ושולפת ומציגה את תכולת הטבלאות Products ו-Suppliers. בבסיס הנתונים של Northwind, כל מוצר מגיע מספק אחד בלבד, אבל כל ספק יכול לספק מוצרים אחדים. יחס זה בין הספק למוצרים נקרא יחס "אחד לרבים". בתרגיל הראשון עליך להשתמש באשפים של Visual Studio 2005 כדי ליצור מקור נתונים (data source) שמתחבר לבסיס הנתונים של Northwind ושולף את תכולת הטבלאות.

צור מקור נתונים

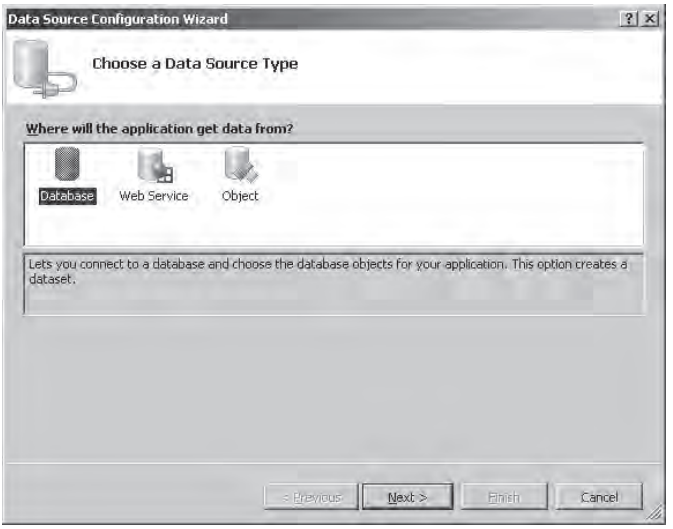
1. באמצעות Visual Studio 2005, צור פרויקט חדש DisplayProducts על פי התבנית Windows Application. שמור את הפרויקט בתיקייה My Documents\Microsoft Press\Visual CSharp Step by Step\Chapter 23.

טיפ

אם אינך זוכר כיצד ליצור יישום Windows חדש, חזור לתרגיל הראשון בפרק 20, "צור את פרויקט ארגון מצלצלי הפעמונים של מידלשייר".

2. בתפריט Data בחר Add New Data Source.

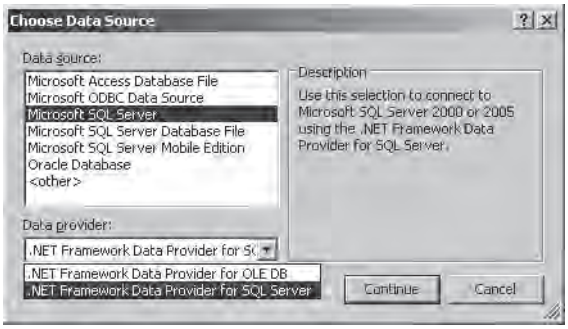
כעת יופעל האשף Data Source Configuration. עליך להשתמש בו כדי ליצור חיבור אל **מקור נתונים** (Data Source), אשר יכול להיות בסיס נתונים, אובייקט או שירות Web. ספר זה אינו עוסק בניצול אובייקטים או שירותי Web כמקורות לנתונים, אולם בפרק 28 תלמד ליצור שירותי Web וגם כיצד להשתמש בהם.



3. בחר בסמל Database ולחץ Next. בעמוד הבא של האשף עליך לספק מידע על החיבור שברצונך ליצור עם בסיס הנתונים. מכיוון שזהו החיבור הראשון שאתה יוצר, לחץ New Connection.

כעת תופיע תיבת הדו-שיח Choose Data Source, ובה תוכל לבחור את מקור הנתונים ואת **ספק הנתונים** (data provider) הרצויים לך. מקור הנתונים הוא סוג של בסיס נתונים שאתה רוצה להשתמש בו, וספק הנתונים הוא אופן החיבור לבסיס הנתונים. יש מקורות נתונים שניתן לגשת אליהם באמצעות ספקים אחדים. לדוגמה, ניתן להתחבר לשרת SQL Server באמצעות NET Framework Data Provider for SQL Server, או באמצעות NET Framework Data Provider for OLE DB. היישום NET Framework Data Provider for SQL Server מותאם במיוחד להתחבר לבסיסי נתונים מסוג SQL Server, בעוד שהיישום NET Framework Data Provider for OLE DB הוא ספק יותר גנרי, ובאמצעותו ניתן להתחבר למקורות נתונים שונים, ולא רק לשרת SQL Server.

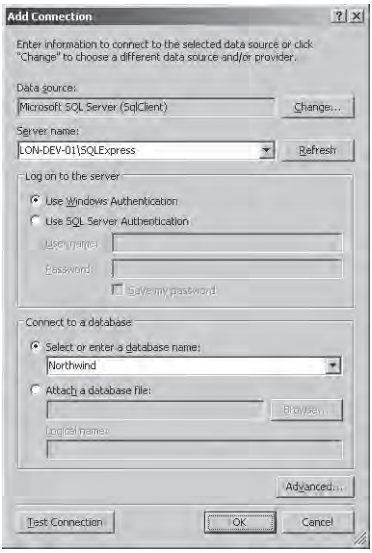
4. עבור יישום זה בחר במקור נתונים Microsoft SQL Server, ובספק נתונים NET Framework Data Provider for SQL Server.



5. לחץ Continue כדי לעבור לשלב הבא.

כעת תוצג תיבת הדו-שיח Add Connection. עליך לציין בה את השרת SQL Server שאתה רוצה להתחבר אליו, את מנגנון האימות (authentication mechanism) שאתה רוצה להפעיל, ואת בסיס הנתונים שאתה רוצה לגשת אליו.

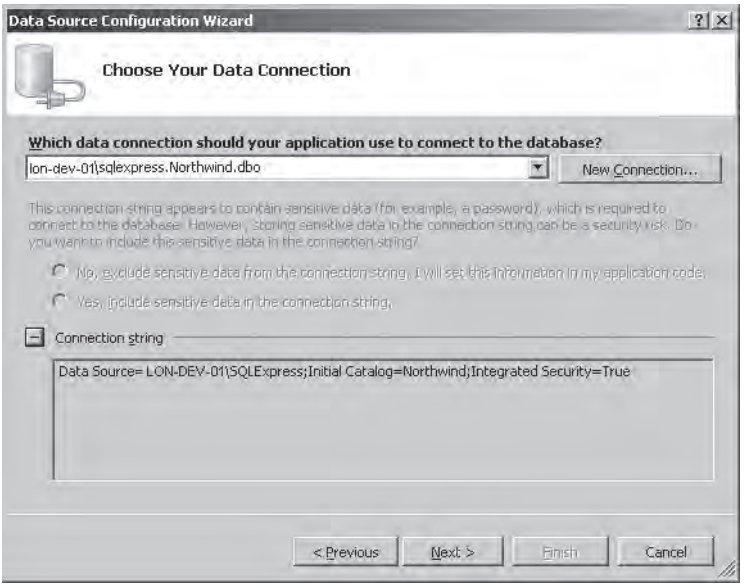
6. בתיבה Server הקלד **YourServer**\SQLExpress, אך במקום **YourServer** הקלד את שם המחשב שלך. בתיבה Log on to the server בחר באפשרות Use Windows Authentication, אשר מנצלת את שם המשתמש שלך ב-Windows, כדי להתחבר לבסיס הנתונים. זו הדרך המומלצת להתחברות לשרתי SQL Server. בחר בבסיס הנתונים Northwind ולחץ OK.



כעת יופיע שוב האשף Data Source Configuration. חיבור הנתונים החדש נקרא **YourServer**\SQLExpress.Northwind.dbo.

7. לחץ על סימן הפלוס (+) הסמוך לתווית Connection String.

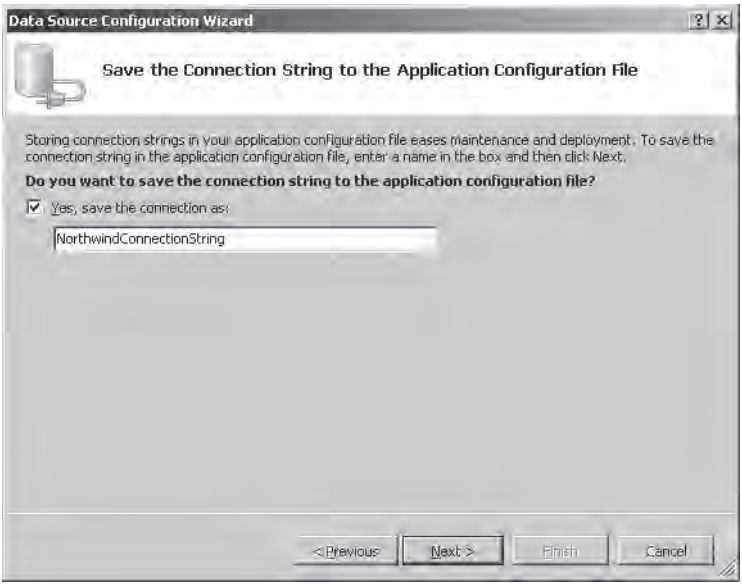
כעת תוכל לראות את המחרוזת המכילה את פרטי ההתחברות שזה עתה בחרת. מידע זה מאוחסן בפורמט שיוכל לשמש את הספק של SQL Server כדי להתחבר לשרת.



8. לחץ Next.

ניתן לשמור את פרטי ההתחברות שזה עתה ציינת גם בקובץ התצורה (configuration file) של היישום. כך תוכל לשנות את מחרוזת ההתחברות מבלי לבנות את היישום מחדש. כל שעליך לעשות, זה לערוך את קובץ התצורה של היישום. הדבר שימושי במיוחד כאשר אתה צופה שבעתיד תצטרך להחליף את בסיס הנתונים המקורי של היישום.

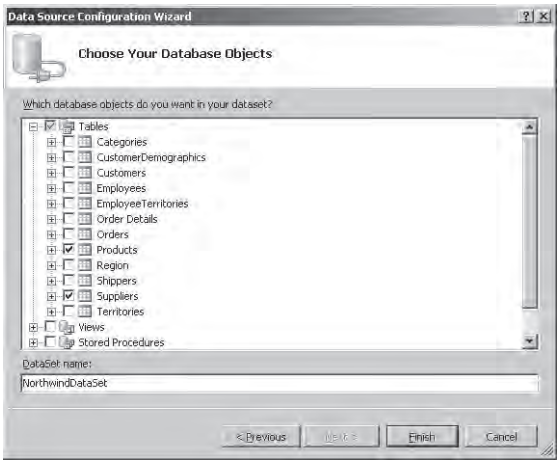
שמור את פרטי ההתחברות לפי שם ברירת המחדל.



9. לחץ Next.

העמוד הבא של האשף מאפשר לבחור את הנתונים שאתה רוצה להשתמש בהם. באפשרותך לאחזר מידע מהטבלאות או מהתצוגות (Views) שבבסיס הנתונים, או לגשת לתוצאות של פרוצדורות ופונקציות שבשרת SQL Server.

10. הרחב את התיקייה Tables וסמן את הטבלה Products ואת הטבלה Suppliers.



כעת האשף יצור אובייקט DataSet בשם NorthwindDataSet שבאמצעותו תוכל לשלוט בנתונים המוחזרים. אובייקט זה מאוחסן בזיכרון ולמעשה, הוא עותק של הטבלאות שנשלפו מבסיס הנתונים.

11. לסיום לחץ Finish.

שימוש בקובץ תצורה של יישום

קובץ תצורה של יישום מאפשר למשתמש לבצע שינויים בחלק מהמשאבים המשמשים את היישום, מבלי לבנות את היישום מחדש. מחרוזת ההתחברות היא דוגמה למשאב שכזה.

כאשר אתה שומר את מחרוזת ההתחברות שחולל האשף Data Source Configuration, מצורף לפרויקט קובץ app.config חדש. זהו המקור של קובץ התצורה של היישום, ומופיע ב- Solution Explorer. לחץ לחיצה כפולה על הקובץ כדי לעיין בתכולתו. כשתעשה זאת, תראה קובץ XML, כמוצג להלן:

```
<?xml version="1.0" encoding="utf-8" ?>
<configuration>
   <configSections>
      <sectionGroup name="userSettings" ... >
         <section name="DisplayProducts.Properties.Settings" ... />
      </sectionGroup>
      <sectionGroup name="applicationSettings" ... >
         <section name="DisplayProducts.Properties.Settings" ... />
      </sectionGroup>
   </configSections>
   <connectionStrings>
      <add name="DisplayProducts.Properties.Settings.
         NorthwindConnectionString"
        connectionString="Data Source=LON-DEV-01\SQLExpress;Initial
Catalog=Northwind;Integrated Security=True"
providerName="System.Data.SqlClient" />
   </connectionStrings>
   <userSettings>
      <DisplayProducts.Properties.Settings />
   </userSettings>
   <applicationSettings>
      <DisplayProducts.Properties.Settings />
   </applicationSettings>
</configuration>
```

מחרוזת ההתחברות מאוחסנת ברכיב <connectionString> של הקובץ. בעת בניית היישום, שדה זה משמש ליצירת קובץ נוסף בשם DisplayProducts.exe.config, שמכיל את אותו המידע בתסדיר זהה, ומאוחסן באותה תיקייה בתור קובץ הפעלה. זהו קובץ התצורה של התוכנית. את הקבצים עליך לפתוח באמצעות קובץ ההפעלה (exe). אם המשתמש צריך להתחבר לבסיס נתונים אחר, הוא יכול לערוך את קובץ התצורה על ידי עורך טקסט ולשנות את התכונה <connectionString> של הרכיב <connectionString>. כאשר היישום יופעל שוב, הוא ישתמש בערך החדש באופן אוטומטי.

האובייקטים DataSet, DataTable ו–DataAdapter

מודל האובייקטים ADO.NET מפעיל אובייקטים מסוג DataSets כדי לאחסן את הנתונים הנשלפים מבסיס הנתונים. יישום אשר מגדיר אובייקטים מסוג DataSets, יכול גם לבצע שאילתות המעבירות את הנתונים אל אובייקט DataSet אחר, ואחר כך להציג או לעדכן את המידע שהוא מכיל. אובייקט מסוג DataSet מכיל אובייקט אחד או יותר מסוג **טבלת נתונים** (DataTable), שכל אחד מהם מייצג את אחת הטבלאות שנבחרו בעת הגדרת האובייקט DataSet. בתרגיל הבא תראה שהאובייקט NorthwindDataSet מכיל שתי טבלאות נתונים (אובייקטים מסוג DataTable) בשמות Products ו–Suppliers.

בבסיס הנתונים Northwind, היחס בין הטבלאות Products ו–Suppliers הוא "אחד לרבים". כל מוצר מסופק על ידי ספק אחד, אך כל ספק יכול לספק מוצרים אחדים ללא הגבלה. בסיס הנתונים Northwind מממש את היחס הזה באמצעות **מפתח ראשי** (primary) ו**מפתח זר** (foreign). האשף Data Source Configuration משתמש במידע זה כדי ליצור **אובייקט קשר** (יחס, relation) DataRelation כחלק מהאובייקט NorthwindDataSet. מטרת האובייקט DataRelation לוודא שהיחס שבבסיס הנתונים מתקיים גם בין טבלאות הנתונים Products ו–Suppliers שנמצאות בזיכרון.

כיצד היישום ממלא את DataSet בערכים בסביבת ההרצה? התשובה לכך נמצאת באובייקט TableAdapter (מתאם טבלאות), אשר מכיל שיטות שניתן להשתמש בהן לבניית אובייקט DataSet. שתי השיטות השימושיות ביותר נקראות Fill ו–GetData. השיטה Fill ממלאה אובייקט DataSet קיים ומחזירה ערך המייצג את מספר השורות שנשלפו. השיטה GetData יוצרת אובייקט DataSet חדש וממלאה אותו בנתונים.

כעת נעבור מהשלב התיאורטי לשלב המעשי.

עיין (Browse) בנתוני המוצרים והספקים

1. בתפריט Data בחר Preview Data.

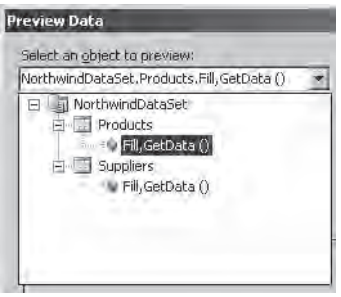
תיבת הדו–שיח Preview Data מוצגת ומאפשרת לצפות בנתונים המוחזרים על ידי מקור הנתונים שייצרת.

טיפ

אם התפריט Preview Data אינו מוצג, לחץ לחיצה כפולה על Form.cs שב- Solution Explorer כדי להציג את Form1 בחלון תצוגת העיצוב.

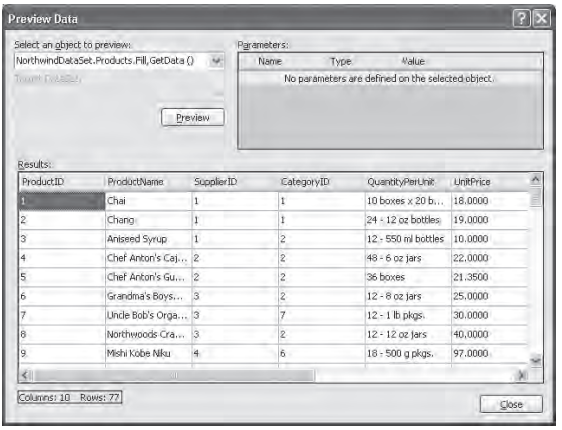
2. לחץ על הרשימה הנפתחת Select an object to preview.

כעת יופיע עץ המכיל את תכולת האובייקט NorthwindDataSet. האובייקט מכיל שני אובייקטים מסוג DataTable שנקראים Products ו–Suppliers. תחת כל טבלת נתונים מופיע צומת שלידו הכיתוב Fill, GetData() (ייתכן שתצטרך להרחיב את טבלת הנתונים Suppliers כדי לראות את הצומת שלה). כל אחד מהצמתים מייצג את האובייקט TableAdapter של כל אחת מהטבלאות.



3. לחץ על הצומת Fill, GetData() שתחת טבלת הנתונים Products, ולחץ על הלחצן Preview.

החלון Results יציג את שורות הטבלה Products שבבסיס הנתונים.



4. ברשימה הנפתחת Select an object to preview, בחר בצומת Fill, GetData() שתחת טבלת הנתונים Suppliers ולחץ Preview.

הפעם יוצגו הנתונים מהטבלה Suppliers.

5. לחץ Close כדי לסגור את תיבת הדו-שיח Preview Data.

הצגת הנתונים ביישום

לאחר שלמדת להגדיר אובייקטים מסוג DataSets, תלמד להשתמש בהם במסגרת יישום.

הצג את נתוני המוצר ביישום Windows Forms

1. ב- Solution Explorer, לחץ לחיצה ימנית על הטופס Form1.cs ושנה את שמו ל-DataForm.cs. כעת תופיע תיבת דו-שיח שתשאל אם אתה רוצה ש- Visual Studio 2005 תשנה את כל ההפניות בפרויקט בהתאם לשם החדש. לחץ Yes.

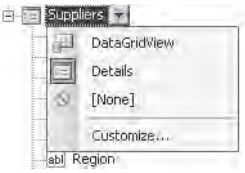
2. הצג את הטופס DataForm בחלון תצוגת העיצוב. היעזר בחלון Properties כדי לשנות את המאפיין Text ל- Suppliers and Products, ושנה את המאפיין Size ל- **"800, 410"** (ללא גרשיים).

3. בתפריט Data בחר Show Data Sources. חלון Data Sources יופיע ויציג את NorthwindDataSet עם טבלאות הנתונים Products ו-Suppliers. לחץ על סימני הפלוס (+) כדי להרחיב את שתי הטבלאות.

כתוצאה יופיעו העמודות שבכל אחת מהטבלאות, ולצד כל אחת יופיע סמל המייצג כיצד היא תוצג ביישום. רוב העמודות יוצגו בתור פקדי textbox, למרות שהעמודה Discontinued שבטבלה Products תופיע בתור תיבת סימון. הסיבה לכך היא, שהעמודה המשויכת בבסיס הנתונים היא עמודת סיביות שיכולה להכיל ערכי True/False בלבד. שים לב גם לכך שהאובייקט Products מופיע פעמיים: פעם אחת כטבלת נתונים עצמאית, ופעם נוספת כטור בטבלה Suppliers. מייד תראה כיצד הדבר מאפשר לשלוט ביחס שבין הספק לבין המוצרים שהוא מספק.

4. בחר בטבלת הנתונים Suppliers. תפריט נפתח יופיע לצד השם. לחץ עליו ובחר Details.

פעולה זו תגרום לשינוי תצוגת הספקים מתצוגת הרשת (grid) של ברירת המחדל לקבוצה של שדות (עם סמל צמוד לטבלה). הפריסה Details (פירוט) נוחה להצגת הנתונים בצד של ה"אחד" ביחס מסוג "אחד לרבים", בעוד שתצוגת הרשת טובה יותר לצד של ה"רבים".

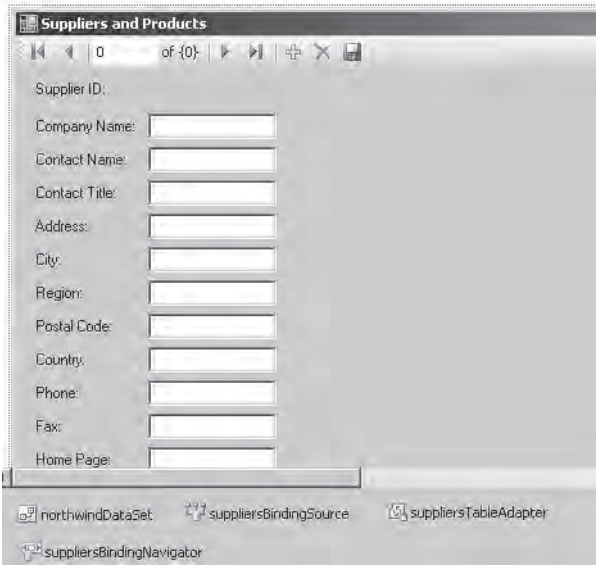


5. בחר בעמודה SupplierID שבאובייקט Suppliers. בתפריט הנפתח יוצגו הדרכים השונות להצגת הנתונים שבעמודה זו. העמודה SupplierID היא למעשה המפתח הראשי (primary key) של הטבלה בבסיס הנתונים, ולכן אין לבצע בה שינויים. על כן, לחץ על הפקד Label.

6. לחץ על האובייקט Suppliers וגרור אותו אל הפינה השמאלית העליונה של הטופס Suppliers and Products.

שים לב שיש רכיבים אחדים תחת הטופס, והם מתוארים בטבלה הבאה. שים לב גם לסרגל הכלים שבראש הטופס, אשר מכיל פריטים המשמשים לעיון ברשימות הספקים; להוספה, עריכה ומחיקה של שורות; וגם לשמירת השינויים בבסיס הנתונים.

רכיב	תיאור
NorthwindDataSet	מקור נתונים זה משמש את הטופס. זהו אובייקט מסוג NorthwindDataSet שמכיל שיטות לעדכון נתונים בבסיס הנתונים.
suppliersBindingSource	רכיב זה משמש לתיווך בין הפקדים שבטופס לבין מקור הנתונים. רכיב מסוג BindingSource עוקב אחר השורה הנוכחית באובייקט DataSet ומוודא שהפקדים בטופס מציגים את הנתונים של השורה. רכיב BindingSource גם מכיל שיטות לדפדוף בין הרשומות באובייקט DataSet תוך כדי הוספה, הסרה ועדכון של שורות.
suppliersTableAdapter	אובייקט מסוג TableAdapter עבור טבלת הספקים, אשר מכיל שיטות לשליפת שורות מטבלת הספקים שבסיס הנתונים כדי למלא איתן את מקור הנתונים.
suppliersBindingNavigator	פקד מסוג bindingNavigator הכולל מנגנון סטנדרטי לעיון בשורות של אובייקט DataSet. הוא למעשה סרגל כלים גלוי שמופיע בראש הטופס וכולל כלים המאפשרים את ביצוע מרבית הפעולות השכיחות הקשורות לנתונים.



טיפ

אם השדות שעל הטופס גבוהים מדי ומסתירים את סרגל הכלים, עליך לגרור את השדה Supplier ID אל המקום הרצוי כאשר השדות עדיין מסומנים, ואז שאר השדות יזוזו איתו.

7. לחץ על האובייקט Products שנמצא בתוך טבלת הנתונים Suppliers וגרור אותו לטופס, מימין לשדה Supplier.

פקד מסוג DataGridView יופיע על הטופס. שים לב להופעת שני רכיבים נוספים תחת הטופס. אחד מהם הוא productsBindingSource – פקד מסוג BindingSource המתאם את השורות שבפקד DataGridView עם האובייקט northwindDataSet. השני הוא productsTableAdapter – המשמש לשליפת שורות מבסיס הנתונים ואחסונן באובייקט Products.

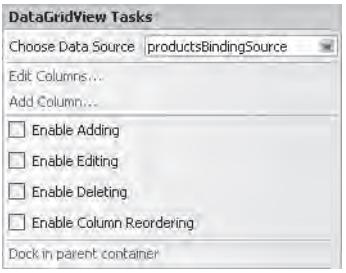
טיפ

חשוב לגרור את האובייקט Products שבטבלת הנתונים Suppliers ולא את האובייקט Products שהינו טבלה עצמאית באובייקט NorthwindDataSet. אם תשתמש באובייקט זה, התצוגה לא תפעל כשורה בסביבת ההרצה. במקום להציג רק את המוצרים של הספק הנבחר, כל המוצגים יוצגו כל הזמן.

8. לחץ על הפקד DataGridView שעל הטופס והרחב אותו כדי שימלא את כל החלק הימני של הטופס.

9. כאשר הפקד DataGridView עודנו מסומן (לחץ עליו אם לא), לחץ על הבקר Smart Tag שבפינה הימנית העליונה של הפקד.

תיבת הדו-שיח Tasks של הפקד DataGridView תוצג. באמצעותה תוכל לשנות את המאפיינים השימושיים ביותר של הפקד DataGridView, ולבצע מטלות כגון שינוי מאפיינים של העמודות המוצגות ושינוי הפעולות שהפקד תומך בהן. כדי לשמור על פשטות היישום, בטל את הסימון בתיבות הסימון Enable Adding, Enable Editing, Enable Deleting ו- Enable Column Reordering. בפרק 24 תלמד עוד על השימוש בפקד DataGridView.



10. הפעל את היישום מבלי לנפות שגיאות. כשהטופס יוצג, תראה את הספק הראשון (Exotic Liquids) וגם את שלושת המוצרים שהוא מספק. לחץ על הלחצן Move next שבסרגל הכלים. כעת יופיע הספק New Orleans Cajun Delights, בשילוב ארבעת המוצרים שהוא מספק. באפשרותך לשנות את פרטי הספק, אבל השינוי לא יישמר בבסיס הנתונים עד אשר תלחץ על הלחצן Save שבסרגל הכלים.

11. לחץ על הלחצן Add שבסרגל הכלים.

הטופס יתאפס ותוכל להזין בו נתונים עבור ספק חדש. שים לב לכך שקוד ספק (supplier ID) חדש נוצר באופן אוטומטי. זכור שנתוני הספק החדש לא יישמרו בבסיס הנתונים עד אשר תלחץ על הלחצן Save שבסרגל הכלים.

12. לחץ על הלחצן Delete שבסרגל הכלים כדי למחוק את הספק החדש.

הספק הקיים (מספר 29) יופיע שוב באופן אוטומטי.

13. סגור את הטופס מבלי לשמור את השינויים, וחזור לסביבת התכנות של Visual Studio 2005.

תכנות של ADO.NET

בתרגילים הבאים תכתוב בעצמך את קוד הגישה לבסיס הנתונים. לא תגרור טבלאות מהחלון Data Sources, אלא תכתוב בעצמך קוד אשר יאפשר את הגישה לבסיס הנתונים. התרגיל נועד לעזור לך ללמוד יותר על ADO.NET ולהבין את מודל האובייקטים של ADO.NET. גם במציאות יהיו מקרים רבים שתצטרך לתכנת או להוסיף קטעי תכנות. שיטת הגרירה של טבלאות משמשת בדרך כלל ליצירת אב-הטיפוס של היישום, אולם ביישום עצמו צריך לעיתים קרובות לעשות יותר מזה ולשלוט באופן פרטני בשליפת הנתונים והטיפול בהם, וזאת ניתן לעשות רק על ידי תכנות מסודר.

היישום שתיצור כעת מחולל דוח פשוט שמציג מידע על הזמנות של לקוח. המשתמש צריך להזין את קוד הלקוח (CustomerID) שעבורו התוכנית תציג את ההזמנות.

חיבור אל בסיס הנתונים

1. צור פרויקט ReportOrders בעזרת התבנית Console Application. שמור אותו בתיקייה My Documents\Microsoft Press\Visual CSharp Step By Step\Chapter 23. לחץ OK.

2. ב- Solution Explorer, שנה את שם הקובץ Program.cs ל-Report.cs.

שים לב ששם המחלקה Program בחלון Code and Text Editor משתנה באופן אוטומטי ל-Report.

3. בחלון Code and Text Editor הוסף תחת המשפט using system.Text; את המשפט הבא:

```
using System.Data.SqlClient;
```

מרחב השמות System.Data.SqlClient מכיל את מחלקות ADO.NET המאפשרות גישה אל השרת SQL Server.

4. אתר את השיטה Main של המחלקה Report. הוסף את המשפט הבא, אשר מכריז על אובייקט מסוג **SqlConnection**:

```
SqlConnection dataConnection = new SqlConnection();
```

SqlConnection היא תת-מחלקה של Connection השייכת ל-ADO.NET, ונועדה לטפל בהתחברות לבסיסי נתונים מסוג SQL Server בלבד.

5. לאחר ההכרזה על המשתנה, הוסף לשיטה Main בלוק try/catch. את הקוד שמטרתו לאפשר גישה לבסיס הנתונים עליך לכתוב בתוך הבלוק try. זכור שעליך להיערך לטיפול בחריגים בכל שימוש שתעשה בבסיס נתונים.

```
try
{
   // You will add your code here in a moment
}
catch(Exception e)
{
  Console.WriteLine("Error accessing the database: " + e.Message);
}
```

6. החלף את ההערה שבבלוק try בקוד הבא, שתפקידו לבצע את ההתחברות לבסיס הנתונים:

```
dataConnection.ConnectionString = "Integrated Security=true;" +
                                  "Initial Catalog=Northwind;" +
                                  "Data Source=YourServer\\
                                   SQLExpress";
dataConnection.Open();
```

חשוב

החלף במאפיין ConnectionString את **YourServer** בשם המחשב שלך או בשם המחשב שמפעיל את SQL Server.

תוכן המאפיין ConnectionString של האובייקט **SqlConnection** זהה לתוכן שחולל האשף Data Source Configuration בשלב 7 של התרגיל "צור מקור נתונים". על פי מחרוזת זו, אימות החיבור אל בסיס הנתונים Northwind שבמופע המקומי של SQL Server Express Edition יבוצע באמצעות Windows Authentication. זו השיטה המומלצת, מכיוון שכך המשתמש אינו צריך (וגם אינו יכול) להקליד שם משתמש או סיסמה כחלק מקוד היישום. שים לב לסימון נקודה-פסיק שמפריד בין רכיבי מחרוזות ההתחברות.

יש פרמטרים נוספים רבים שניתן לצרף למחרוזת ההתחברות. תוכל ללמוד עליהם אם תעיין ב- MSDN Library for Visual Studio 2005.

נוהל האימות של SQL Server

Windows Authentication שימושי ביותר כאשר מאמתים משתמשים מה-domain של Windows. אולם, ייתכנו מקרים שלמשתמש שניגש לבסיס הנתונים אין חשבון Windows, כמו למשל ביישומים עבור משתמשים דרך האינטרנט. במקרים כאלה, צריך להשתמש בפרמטרים User ID ו-Password, באופן הבא:

```
string userName = ...;
string password = ...;
// Prompt the user for their name and password, and fill these
variables

myConnection.ConnectionString = "User ID=" + userName +
";Password=" + password + ";Initial Catalog=Northwind;Data
Source=YourServer\\SQLExpress";
```

העצה הבאה חשובה ביותר: לעולם אל תקבע את שם המשתמש ואת הסיסמה בתוך קוד היישום. אם תעשה זאת, כל מי שישיג עותק של קוד המקור (או שיפצח את קוד ההרצה) יוכל לראות את המידע הזה, כלומר השם והסיסמה, והדבר עלול לפגוע באבטחה, אם הדבר חשוב לך!

השלב הבא הוא לבקש מהמשתמש להזין קוד לקוח (CustomerID) ואחר כך לפנות בשאילתה אל בסיס הנתונים כדי לאתר את ההזמנות שלו.

חקור את בסיס הנתונים

1. הוסף את המשפטים הבאים לאחר המשפט dataConnection.Open();:

```
Console.Write("Please enter a customer ID (5 characters): ");
string customerId = Console.ReadLine();
```

משפטים אלה מאפשרים למשתמש להזין את קוד הלקוח ושומרים אותו במשתנה המחרוזת customerID.

2. הוסף את המשפטים הבאים לאחר המשפטים שזה עתה כתבת:

```
SqlCommand dataCommand = new SqlCommand();
dataCommand.Connection = dataConnection;
dataCommand.CommandText =
   “SELECT OrderID, OrderDate, “ +
   “ShippedDate, ShipName, ShipAddress, “ +
   “ShipCity, ShipCountry “;
dataCommand.CommandText +=
   “FROM Orders WHERE CustomerID=’” +
   customerId + “’”;
Console.WriteLine(“About to execute: {0}\n\n”, dataCommand.
CommandText);
```

המשפט הראשון יוצר אובייקט מסוג SqlCommand. בדומה ל-SqlConnection, זו גרסה ייעודית של אחת ממחלקות ADO.NET, שנקראת Command, המעוצבת במיוחד עבור גישה לשרת SQL Server. אובייקט מסוג Command משמש להפעלת פקודות הפועלות על מקור נתונים. אם המקור הוא relational database, הפקודה היא משפט SQL.

המשפט השני בקוד אחראי לקביעת המאפיין Connection של האובייקט SqlCommand עבור ההתחברות שזה עתה יצרת. שני המשפטים הבאים ממלאים את המאפיין CommandText בנתונים באמצעות המשפט SQL SELECT אשר מאחזר נתונים מטבלת ההזמנות על פי קוד הלקוח שמופיע במשתנה customerID (ניתן לעשות זאת גם במשפט אחד, אך עדיף לעשות זאת בשני משפטים, מכיוון שהדבר מקל על הבנת הקוד). המשפט Console.WriteLine מדפיס על המסך את הפקודה שעומדת להתבצע.

3. הוסף את המשפט הבא מייד לאחר הקוד שכתבת:

```
SqlDataReader dataReader = dataCommand.ExecuteReader();
```

הדרך המהירה ביותר לאחזר נתונים מבסיס נתונים מסוג SQL Server, היא להשתמש במחלקה SqlDataReader. מחלקה זו מאחזרת שורות מבסיס הנתונים במהירות המרבית המתאפשרת על ידי הרשת, ומצרפת אותן ליישום.

משימתך הבאה היא לסקור בפעולה איטרטיבית את ההזמנות (אם יש כאלו) ולהציגן. שים לב שזהו אחזור בלבד (קריאה) ולא פעולת עדכון!

קריאת נתונים והצגת הזמנות

1. הוסף את הלולאה while שלהלן מייד לאחר המשפט שיוצר את האובייקט SqlDataReader:

```
while (dataReader.Read())
{
   // Code to display the current row
}
```

השיטה Read של המחלקה SqlDataReader מאחזרת את השורה הבאה מבסיס הנתונים אל היישום. היא מחזירה ערך true כאשר שורה נוספת הועברה בהצלחה; אחרת, היא מחזירה ערך false. לולאת while זו קוראת שורות מהמשתנה dataReader עד אשר אין יותר שורות לקרוא.

2. הוסף את המשפטים הבאים לגוף הלולאה while שיצרת בשלב הקודם:

```
int orderId = dataReader.GetInt32(0);
DateTime orderDate = dataReader.GetDateTime(1);
DateTime shipDate = dataReader.GetDateTime(2);
string shipName = dataReader.GetString(3);
string shipAddress = dataReader.GetString(4);
string shipCity = dataReader.GetString(5);
string shipCountry = dataReader.GetString(6);
Console.WriteLine(
    “Order {0}\nPlaced {1}\nShipped {2}\n” +
    “To Address {3}\n{4}\n{5}\n{6}\n\n”, orderId, orderDate,
    shipDate, shipName, shipAddress, shipCity, shipCountry);
```

תהליך זה משמש לקריאת נתונים מהאובייקט SqlDataReader. אובייקט מסוג SqlDataReader מכיל את השורה האחרונה שאוחזרה מבסיס הנתונים. תוכל להשתמש בשיטות GetXXX השונות כדי לאחזר את הנתון שבכל אחת מהעמודות שבשורה - לכל אחד מסוגי הנתונים השכיחים יש שיטה GETXXX נפרדת. לדוגמה, כדי לקרוא ערך int, עליך להשתמש בשיטה GetInt32, וכדי לקרוא מחרוזת עליך להשתמש בשיטה GetString. תוכל בוודאי לנחש בקלות איזו שיטה משמשת כדי לקרוא ערך מסוג DateTime. שיטות GetXXX השונות מקבלות כפרמטר את העמודה שעליהן לקרוא, כאשר 0 היא העמודה הראשונה, 1 היא השנייה וכן הלאה. מטרת הקוד שלעיל לקרוא את העמודות השונות שבשורת ההזמנות (Orders) הנוכחית, לאחסן את הערכים בקבוצה של משתנים, ולבסוף - להדפיס את ערכי המשתנים על המסך.

Firehose Cursors

נעילת נתונים (data locking) היא אחד החסרונות העיקריים ביישומים שמפעילים בסיסי נתונים מרובי משתמשים. למרבה הצער, לעיתים קרובות יש יישומים שנועלים שורות כאשר הם מאחזרים אותן ממסד נתונים, כדי למנוע ממשתמשים אחרים לשנות את הנתונים כל עוד הם ברשותם. במקרים קיצוניים היישומים מונעים ממשתמשים אחרים אפילו קריאה של הנתונים שננעלו. אם היישום שולף מספר רב של שורות, הוא נועל למעשה חלק גדול מהטבלה. כאשר משתמשים רבים מפעילים את אותו יישום או פונים אל בסיס הנתונים באותו הזמן, הם יצטרכו להמתין זמן רב עד לשחרור הנעילות, והדבר יגרום להאטת הפעילות של כל המערכת ולשיבוש העבודה. אחד היעדים של המחלקה SqlDataReader, אשר עוסקת בקריאת נתונים בלבד, הוא לפתור את הבעיה הזו. המחלקה מאחזרת שורה אחת בכל פעם אך אינה נועלת אותה מפני אחרים. הדבר משפר במידה רבה את האפשרות של יישומים אחרים לפעול ללא הפרעה. המחלקה SqlDataReader נקראת לעיתים "firehose cursor" (cursor זה ראשי התיבות של current set of rows, ובעברית - רצף השורות הנוכחי). ביצועי המחלקה SqlDataReader טובים יותר מאלה של DataSet, גם במקרים של אחזור נתונים בלבד. אובייקטים מסוג DataSet עושים שימוש ב-XML כדי לייצג את הנתונים שהם מאחסנים. גישה זו אמנם מאפשרת גמישות וכל אחד מהרכיבים אשר משמשים את המחלקה DataSet ויכולים לקרוא פורמט XML, יכולים גם לעבד נתונים. המחלקה SqlDataReader עושה שימוש בפורמט העברת הנתונים של SQL Server כדי לאחזר נתונים ואינה משתמשת בפורמט ביניים כגון XML. אולם, ההתייעלות הזו באה על חשבון הגמישות הקיימת בעת שימוש במחלקה DataSet.

חשוב

בפעולות עדכון, שלא נדון בהן כאן, חייבים לנעול את השורות כדי לא לגרום לשיבוש של בסיס הנתונים. הבה נסביר את הבעיה על ידי דוגמה: נניח שהערך של שדה המלאי הוא 10. משתמש א' קורא ומעדכן ל-15 כי התקבלה סחורה. משתמש ב' קורא גם הוא את הערך 10 בשעה שא' עסק בחישוב הכמות ומעדכן על ידי הוספה של 2, ולאחר שא' כותב את העדכון שלו הוא כותב 12. התוצאה הסופית שגויה, כי אנו רוצים שסך הכל יהיו במלאי 17 פריטים! על כן, כאשר א' קורא את המלאי, הוא חוסם את הגישה עד לסיום העדכון שלו וכתיבת 15. כעת, ב' שהמתין יקרא 15 ויעדכן ל-17 - וזה בדיוק מה שצריך להיות!

בסיום העבודה עם בסיס הנתונים, רצוי תמיד לשחרר את המשאבים שבהם משתמשים.

התנתקות מבסיס הנתונים

1. בחלונית Code, הוסף את המשפט הבא מייד לאחר הלולאת while:

```
dataReader.Close();
```

משפט זה סוגר את האובייקט SqlDataReader. חשוב תמיד לסגור אובייקטים מסוג SqlDataReader בסיום השימוש בהם, מכיוון שרק לאחר שעושים זאת ניתן להשתמש באובייקט SqlConnection הנוכחי כדי להפעיל פקודות נוספות. מומלץ שתעשה זאת גם אם כל מה שאתה מתכוון לעשות לאחר מכן זה סגירת האובייקט SqlConnection.

2. הוסף את הבלוק finally הבא מייד לאחר הבלוק catch:

```
finally
{
   dataConnection.Close();
}
```

החיבורים לבסיסי נתונים הם משאבים חשובים. עליך לוודא שהם אכן נסגרים כאשר אתה מסיים להפעיל אותם. הוספת משפט זה לבלוק **finally** תגרום לסגירת האובייקט **SqlConnection** גם במקרה של חריג. זכור שהקוד בבלוק **finally** מופעל תמיד עם סיום פעולת המטפל catch.

3. בתפריט Debug בחר Start Without Debugging כדי לבנות ולהפעיל את היישום.

על פי ברירת המחדל, הפקודה Start Without Debugging מפעילה את התוכנית, ולבסוף סוגרת את חלון התצוגה (Console) רק לאחר קבלת אישור, כי היא ממתינה לכך שתקרא את הפלט. אם תבחר בפקודה Start Debugging, חלון התצוגה ייסגר מייד בסיום התוכנית ולא תוכל לקרוא את הפלט.

4. כאשר תתבקש להזין customer ID, הקלד VINET והקש Enter. כתוצאה יופיע המשפט SQL SELECT, ולאחריו ההזמנות שהתקבלו מהלקוח. ניתן לגלול את מסך התצוגה מעלה ומטה כדי לראות את כל הנתונים. לסיום הקש Enter כדי לסגור את חלון התצוגה.

5. הפעל שוב את היישום, אך הקלד BONAP כשתתבקש להזין את קוד הלקוח.

יוצגו שמות אחדים, אך לאחריהן תופיע הודעת שגיאה: "Error accessing the data base. Data is Null.". הבעיה היא שבסיסי נתונים טבלאיים מאפשרים לחלק מהעמודות להכיל ערכי null. ערך null דומה במקצת למשתנה null בשפת C#: אין לו ערך, ואם תנסה להשתמש בו, תופיע הודעת שגיאה. בטבלת ההזמנות, העמודה ShippedDate תכיל null כל עוד ההזמנה לא נשלחה.

6. הקש Enter כדי לסגור את חלון התצוגה.

סגירת חיבור

בחלק גדול מהיישומים הישנים יש נטייה להתחבר אל בסיס הנתונים כאשר היישום מופעל ולא להתנתק עד אשר היישום מסיים את פעולתו. ההיגיון שעמד מאחורי שיטת עבודה זו היה, שההתחברות וההתנתקות הינם תהליכים גוזלי משאבים וגוזלי זמן. שיטת העבודה הזו פגעה ביכולת יישומים אחדים לפעול במקביל על אותו בסיס נתונים, מכיוון שכל אחד מהם היה מחובר למשך זמן רב ליישום שפתח אותו ולא איפשר לאחרים גישה, גם כאשר המשתמש נמצא בהפסקת הצהרים שלו, למשל. רוב בסיסי הנתונים מגבילים את מספר החיבורים שהם יכולים לקבל בו-זמנית. לעיתים הסיבות להגבלה קשורות לרשיון התוכנה, אך לרוב הסיבה העיקרית היא שכל חיבור גוזל כמות מסוימת של משאבים מהשרת של בסיס הנתונים, שאינם אינסופיים. כמובן שבסופו של דבר, מספר החיבורים המקבילים מגיע למגבלה של אותו בסיס נתונים.

רוב ספקי הנתונים של NET Framework, וביניהם גם SQL Server, כוללים מנגנון **שיתוף חיבורים** (**connection pooling**). חיבורים לבסיסי נתונים נוצרים ומאוחסנים במאגר (pool). כאשר יישום כלשהו זקוק לחיבור, ספק הגישה לנתונים שולף חיבור זמין מהמאגר. כאשר היישום סוגר את החיבור, החיבור מוחזר למאגר שבו הוא זמין ליישום הבא שיהיה זקוק לו. משמעות הדבר היא שהפתיחה והסגירה של החיבור לבסיס הנתונים אינן גוזלות משאבים רבים, מכיוון שהן זמינות בדרך אחרת: כדי לפתוח חיבור צריך לשלוף חיבור זמין מהמאגר, וסגירת החיבור אינה מחייבת התנתקות מבסיס הנתונים, אלא רק החזרת החיבור למאגר. כתוצאה מגישה זו, עליך להחזיק בחיבורים רק בעת השימוש בהם - פתח את החיבור כאשר אתה צריך אותו וסגור אותו כאשר אתה מסיים לעבוד איתו.

שים לב שהשיטה ExecuteReader מהמחלקה SqlCommand, אשר יוצרת אובייקט מסוג SqlDataReader, היא שיטה מועמסת. ניתן להעביר לה פרמטר System.Data.CommandBehavior אשר סוגר באופן אוטומטי את החיבור שמשמש את האובייקט SqlDataReader כאשר האובייקט נסגר. לדוגמה:

```
SqlDataReader dataReader =
   dataCommand.ExecuteReader(System.Data.CommandBehavior.
   CloseConnection);
```

כאשר אתה קורא את הנתונים מהאובייקט SqlDataReader, עליך לוודא שאין ביניהם נתונים שערכם null. בתרגיל האחרון בפרק זה עליך להוסיף ליישום ReportOrder משפט שמטרתו לבדוק אם יש בנתונים ערכי null.

טיפול בערכי null שבבסיס הנתונים

1. בחלון Code and Text Editor, אתר את הלולאה while שמדפדפת בשורות שנשלפו באמצעות המשתנה dataReader. שנה את גוף הלולאה באופן הבא:

```
while (dataReader.Read())
{
   int orderId = dataReader.GetInt32(0);
   if (dataReader.IsDBNull(2))
   {
      Console.WriteLine("Order {0} not yet shipped\n\n",
      orderId);
   }
   else
   {
      DateTime orderDate = dataReader.GetDateTime(1);
      DateTime shipDate = dataReader.GetDateTime(2);
      string shipName = dataReader.GetString(3);
      string shipAddress = dataReader.GetString(4);
      string shipCity = dataReader.GetString(5);
      string shipCountry = dataReader.GetString(6);
      Console.WriteLine(
         "Order {0}\nPlaced {1}\nShipped {2}\n" +
         "To Address {3}\n{4}\n{5}\n{6}\n\n", orderId,
          orderDate,
         shipDate, shipName, shipAddress, shipCity,
         shipCountry);
   }
}
```

משפט זה מפעיל את השיטה IsDBNull כדי לקבוע אם ערך העמודה ShippedDate (עמודה 2 בטבלה) הוא null. אם הערך הוא null, לא יבוצע ניסיון לאחזר אותו וגם לא עמודות אחרות, מכיוון שאם אין תאריך משלוח גם הערך שלהם יהיה null. אם ערך העמודה ShippedDate אינו null, גם שאר העמודות ייקראו ויודפסו על המסך כמו קודם.

2. בצע הידור של היישום והפעל אותו. הקלד BONAP כאשר תתבקש להזין את שם המשתמש. הפעם לא תהיינה הודעות שגיאה, ולפניך תוצג רשימת הזמנות שטרם נשלחו.

☯ **אם ברצונך להמשיך לפרק הבא**

השאר את Visual Studio 2005 פעילה ועבור לפרק 24.

☯ **אם ברצונך לכבות כעת את Visual Studio 2005**

פתח את תפריט Menu ולחץ Exit. אם מופיעה תיבת דו-שיח Save, לחץ Yes.

פרק 23 – טבלה מסכמת

המשימה	צריך
ליצור חיבור לבסיס נתונים באופן גרפי ב- Visual Studio 2005.	להשתמש באשף Data Source Configuration. האשף יבקש ממך להזין את פרטי ההתחברות כדי ליצור את האובייקטים המשמשים לעבודה עם בסיס הנתונים.
לעיין בנתונים ב- Visual Studio 2005.	בתפריט Data יש בחר Preview Data. תיבת הדו-שיח PreviewData תוצג ותאפשר לבחור באובייקטים DataSet ו-DataTable הרצויים לך. כדי לצפות בנתונים לחץ Preview.
לשנות את אופן הצגת טבלת הנתונים על הטופס.	בחלון Data Source (אם החלון אינו מופיע, לחץ Show Data Source בתפריט Data) לחץ DataTable ובחר בסגנון הרצוי (DataGridView, Details או None) מהתפריט הנפתח.
לשנות את אופן הצגת השדות שבמקור המידע על פני הטופס.	בחלון Data Source, הרחב את טבלת הנתונים המציגה את השדות שיש לשנות. כעת לחץ על השדות ובחר את אופן הצגתם מתוך התפריט הנפתח שיופיע.
להוסיף DataTable לטופס.	גרור את האובייקט DataTable מחלון Data Sources לטופס. כתוצאה יתווסף לטופס אוסף של שדות או DataGridView. סביבת הפיתוח תחולל אובייקטים מסוג DataSet, BindingSource, TableAdapter ו-BindingNavigator כדי לחבר בין שדות הטופס או הפקד DataGridView לבין בסיס הנתונים וכדי לעיין במידע.
להתחבר לבסיס נתונים באמצעות תכנות.	צור אובייקט חדש מסוג **SqlConnection**, קבע את המאפיין **ConnectionString** שלו, הגדר באמצעותו את בסיס הנתונים שיש להשתמש בו, וקרא לשיטה **Open**.
ליצור ולהפעיל שאילתה על בסיס הנתונים באמצעות קוד.	צור אובייקט מסוג **SqlCommand**. הצב אובייקט תקני מסוג **SqlConnection** במאפיין **Connection** של האובייקט החדש. קרא לשיטה **ExecuteReader** כדי שתבצע את השאילתה ותיצור אובייקט מסוג **SqlDataReader**.
לאחזר מידע באמצעות אובייקט מסוג **SqlDataReader**.	באמצעות השיטה **IsDBNull**, ודא שערך הנתון אינו null. אם הערך אינו null, השתמש בשיטה **GetXXX** המתאימה (כגון **GetInt32** או **GetString**) כדי לאחזר את הנתונים.

פרק 24

כריכת נתונים והאובייקט DataSet

כאשר תסיים פרק זה, תוכל:

- לכרוך מאפיין של פקד אל מקור נתונים בסביבת העיצוב או בסביבת ההרצה, על ידי כריכה פשוטה (simple binding).
- לכרוך פקד לרשימת משתנים ממקור הנתונים על ידי כריכה מורכבת (complex binding).
- לעצב מחלקה DataSet שמכילה טבלאות (DataTables) ומתאמי טבלאות (TableAdapter).
- לכתוב קוד שנועד ליצור מופע של DataSet ולמלא אותו בערכים.
- להשתמש בפקד DataGridView כדי לשנות את הנתונים שבאובייקט DataSet.
- לבצע בדיקות תקינות לשינויים אשר המשתמש ביצע באמצעות הפקד DataGridView.
- לעדכן בבסיס הנתונים את השינויים שבוצעו באובייקט DataSet על ידי האובייקט DataAdapter.

בפרק 23 למדת להשתמש ב-ADO.NET של Microsoft לביצוע שאילתות ולעדכון בסיסי נתונים. בעזרת **DataSet**, אחזרת נתונים מטבלאות הספקים והמוצרים והצגת אותם בטופס הראשי. באמצעות האשף Data Source Configuration קבעת את תצורת מחלקות **DataTable** השונות, ועל ידי גרירת טבלאות הנתונים השונות מ**בסיס הנתונים** אל **טופס**, יצרת את הפקדים **DataSet**, **BindingSource**, **TableAdapter** ו-**BindingNavigator**. לצורך פעולות אלו לא נזקקת לכתיבת קוד בעצמך. בפרק זה, תלמד לכתוב משפטים בשפת #C שמטרתם ביצוע פעולות כאלו. כך תבין את התהליכים שעושה עבורך סביבת הפיתוח Visual Studio 2005, וגם תוכל לכתוב יישומים מורכבים יותר שיכולים לבצע מספר רב יותר של פעולות.

בפרק זה תלמד רבות על **כריכת הנתונים** (data binding) - יצירת קשר בין מאפיין של פקד ומקור הנתונים. תלמד כיצד לכרוך באופן דינמי מאפיינים של פקדים אל אובייקט **DataSet** על ידי כריכת נתונים פשוטה. תלמד גם כיצד להשתמש בכריכת נתונים מורכבת עבור הפקדים **ComboBox** ו-**ListBox**. תלמד על עיצוב ושימוש במחלקות **DataSet** באמצעות הקוד. ובעיקר, תלמד להשתמש באובייקטים מסוג **DataSet** בשילוב עם הפקד **DataGridView** כדי לעדכן ביעילות את בסיס הנתונים.

פקדי Windows Forms וכריכת נתונים

את רוב המאפיינים, של חלק גדול מפקדי Windows Forms, ניתן לשייך (או לכרוך) אל מקור נתונים. לאחר הכריכה, הערך שבמקור הנתונים משנה את ערך המאפיין הכרוך ולהפך. כבר ראית את תהליך הכריכה בפעולה כאשר השתמשנו בפקדים **TextBox** ו-**DataGridView** בפרויקט DisplayProducts בפרק 23. **הפקדים** (controls) שעל הטופס נכרכו לאובייקטים מסוג **BindingSource** השייכים לאובייקט **DataSet**, שמכיל את הרישומים מהטבלה Suppliers והטבלה Products אשר בבסיס הנתונים.

הפקדים של Windows Forms תומכים בשני סוגי כריכה: פשוטה ומורכבת. **כריכת נתונים פשוטה** (**simple data binding**) מאפשרת לשייך מאפיין של פקד או טופס אל **אחד** הערכים שבמקור המידע; **כריכת נתונים מורכבת** (**complex data binding**) משמשת לשיוך הפקד כולו אל רשימת ערכים. כריכת נתונים פשוטה מתאימה לפקדים מסוג **TextBox** או Label המכילים ערך בלבד. כריכת נתונים מורכבת מתאימה יותר עבור פקדים שכוללים מספר רב של ערכים, כגון **ListBox**, **ComboBox** או **DataGridView**.

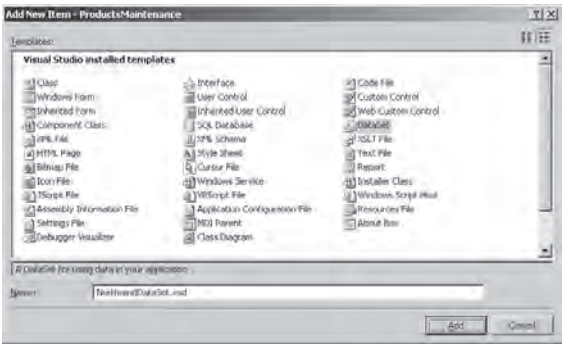
הגדרת DataSet וכריכת נתונים פשוטה

כריכת נתונים פשוטה מתאימה להצגת ערך בודד ממקור הנתונים, אשר יכול להיות אחד מאלה, למשל: תא יחיד באובייקט מסוג **DataSet**, ערך המאפיין של פקד אחר או משתנה פשוט. ניתן לבצע כריכת נתונים פשוטה בסביבת העיצוב באמצעות המאפיין **DataBinding** של הפקד. בתרגיל הבא עליך להגדיר **DataSet** חדש שמגדיר מקור נתונים שמחזיר שורה בודדת, ואחר כך לכרוך את המאפיין **Text** של הפקד **Label** אל אובייקט **BindingSource** של אותו **DataSet**.

הגדר מחלקת DataSet

1. ב- Visual Studio 2005, היעזר בתבנית Windows Application כדי ליצור פרויקט ProductsMaintenance בתיקייה My Documents\Microsoft Press\Visual CSharp Step by Step\Chapter 24.
2. בתפריט Project בחר Add New Item.

בתיבת הדו-שיח Add New Item שתופיע, יוצגו תבניות שונות של אובייקטים שניתן להוסיף לפרויקט.

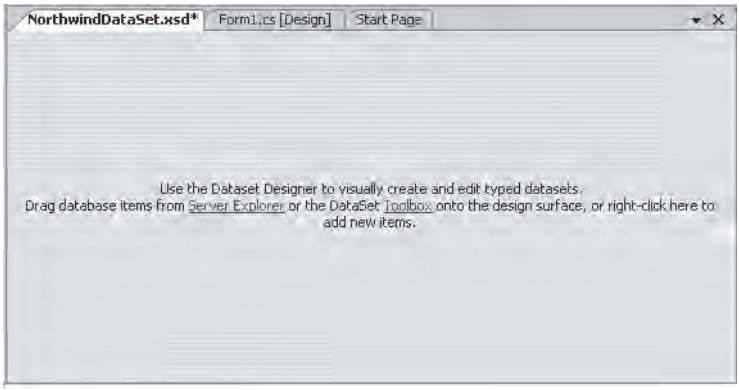


3. בתיבת הדו-שיח Add New Item בחר בתבנית DataSet, הקלד **NorthwindDataSet.xsd** בשדה Name ולחץ Add.

הערה

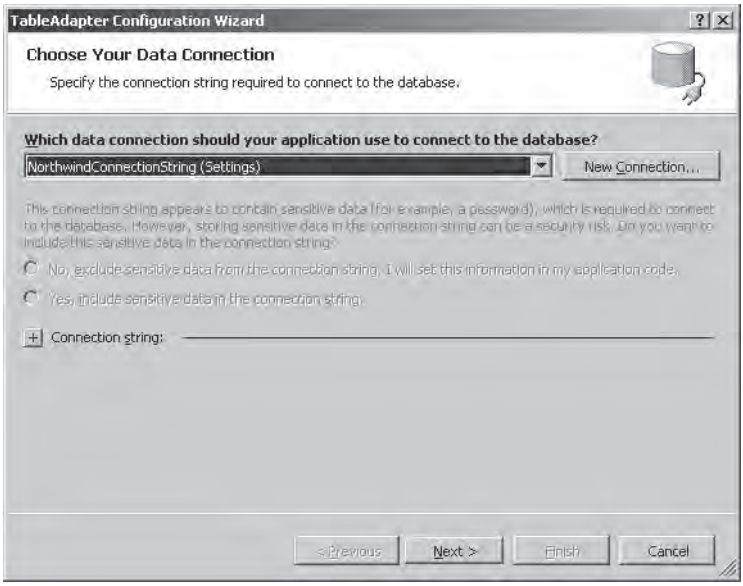
יש תמיד ליצור את ההגדרות של DataSet בקובץ עם סיומת "xsd". ההגדרות של DataSet הן למעשה סכמות (מבנה) של XML, שסביבת הפיתוח משתמשת בהן כדי לחולל את הקוד בעת בניית היישום.

כעת יופיע חלון DataSet Designer.



4. בערכת הכלים הרחב את הקטגוריה DataSet אם דרוש, ולחץ על TableAdapter. לחץ על נקודה כלשהי בחלון DataSet Designer.

פעולה זו תגרום להוספת אובייקטים מסוג DataTable ו-DataAdapter לחלון DataSet Designer, ולהופעת האשף TableAdapter Configuration.



5. בדף Choose Your Data Connection שבאשף TableAdapter Configuration לחץ Next.

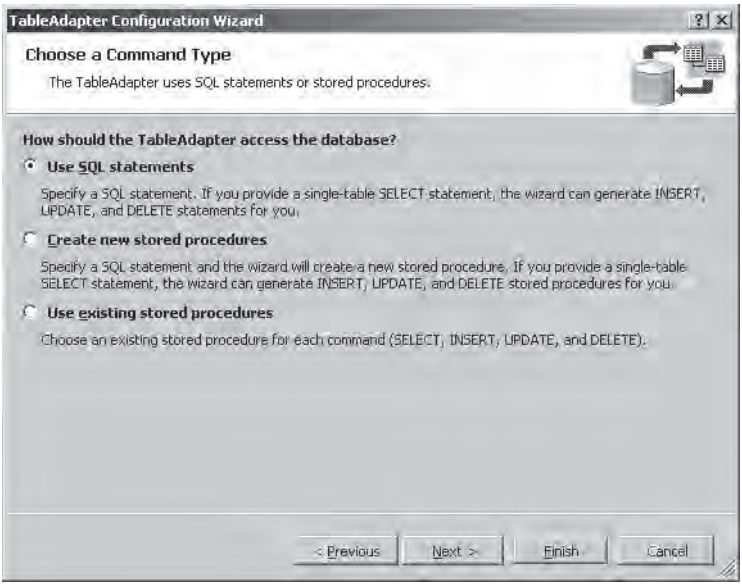
הערה

בעמוד Choose Your Data Connection שבאשף TableAdapter Configuration סוג ההתחברות לנתונים יהיה **NorthwindConnectionString** או **YourServer\ sqlexpress.Northwind.dbo**. הדבר תלוי אם סביבת הפיתוח נשארה פעילה מאז ביצוע התרגילים בפרק 23, או שסגרת ופתחת אותה שוב. בכל מקרה, את התרגיל הבא ניתן לבצע עם כל אחד משני החיבורים הללו.

6. אם מופיע העמוד "Save the connection string to the application configuration file", שמור את מחרוזת ההתחברות תחת השם NorthwindConnectionString ולחץ Next.

כעת יופיע העמוד Choose a Command Type.

7. בעמוד "Choose a Command Type" עליך לציין כיצד האובייקט TableAdapter צריך לגשת לבסיס הנתונים. תוכל להגדיר בעצמך משפטי SQL; לגרום לאשף לחולל פרוצדורות מובנות לכימוס (encapsulate) של משפטי SQL; או להשתמש בפרוצדורות קיימות. בחר Use SQL statements ולחץ Next.



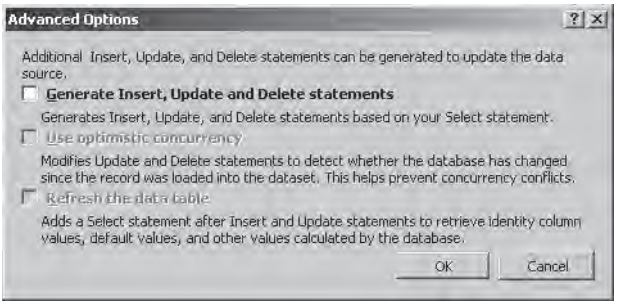
כעת יוצג העמוד Enter SQL Statement.

8. משפט SQL הבא מחשבים את מספר השורות בטבלה Products. הקלד אותו.

```
SELECT COUNT(*) AS NumProducts
FROM Products
```

9. לחץ Advanced Options.

כתוצאה תוצג תיבת הדו-שיח Advanced Options.



בנוסף ליכולת האובייקט **TableAdapter** לאחזר נתונים מבסיס הנתונים, הוא גם יכול להוסיף, לעדכן ולמחוק שורות. האשף יחולל משפטי SQL של INSERT, UPDATE ו-DELETE באופן אוטומטי, על פי הטבלה שבחרת עבור המשפט SQL SELECT (במקרה שלנו, זו הטבלה Products).

10. משפט SQL SELECT המשמש להעברת נתונים מונה את מספר השורות בטבלה Products. אין צורך לחולל משפטי SQL עבור INSERT, UPDATE ו-DELETE, כי לא ניתן לעדכן ערך זה באופן ישיר. בטל את הסימון של "Generate Insert, Update, and Delete statements" ולחץ OK.

11. באשף TableAdapter Configuration לחץ Next.

כעת תוצג תיבת הדו-שיח "Choose Methods to Generate".

12. אובייקט מסוג **TableAdapter** יכול לחולל שתי שיטות בכדי להציב באובייקט **DataTable** את השורות שאוחזרו מבסיס הנתונים: השיטה **Fill** אשר מקבלת כפרמטר **DataTable** או **DataSet** קיימים וממלאה אותם בנתונים, והשיטה **GetData** אשר יוצרת אובייקט **DataTable** חדש ומזינה אותו בנתונים. ודא ששתי השיטות נבחרו ולחץ Next.

13. האשף משתמש במידע שהזנת כדי לחולל מחלקה TableAdapter חדשה. לחץ Finish כדי לסגור את האשף.

המחלקה **DataTable1TableAdapter** מסוג **TableAdapter**, והמחלקה **DataTable1** מסוג DataTable יופיעו בחלון DataSet Designer.

14. בחלון DataSet Designer, לחץ על **DataTable1**. באמצעות חלון Properties שנה את המאפיין **Name** ל-**NumProductsTable**. לחץ על **DataTable1TableAdapter** ושנה את המאפיין Name ל-**NumProductsTableTableAdapter**. האובייקטים שבחלון DataSet Designer נראים כעת כך:

שים לב שהטבלה **NumProductsTable** מכילה **עמודה** אחת בשם **NumProducts**. כאשר המשפט SQL SELECT פועל. בעמודה זו מוצב ערך הביטוי NumProducts, אשר מייצג את מספר השורות בטבלה Products.

15. בתפריט Build בחר Build Solution.

כעת ייווצרו הקוד והאובייקטים של האובייקט DataSet שנשתמש בהם בתרגיל הבא.

טיפ

כדי לצפות בקוד שנוצר על ידי Visual Studio 2005 עליך להרחיב את NorthwindDataSet.xsd שב- Solution Explorer, ואז ללחוץ לחיצה כפולה על הקובץ NorthwindDataSet.Designer.cs. הקפד לא לשנות את הקוד.

בתרגיל הבא עליך לכרוך (bind) את הערך אשר מעביר המתאם **NumProductsTableAdapter** מבסיס הנתונים אל טבלת הנתונים **NumProducts** שבאובייקט **NorthwindDataSet**, אל המאפיין **Text** של הפקד **Label**. כאשר היישום פועל, התווית תציג את מספר השורות בטבלה Products.

כריכה לעמודה בטבלת הנתונים

1. הצג את **Form1** בחלון תצוגת העיצוב. ב- Solution Explorer, שנה את שם הקובץ מ-Form1.cs ל-ProductsForm.cs. אם תוצג שאלה בקשר לעדכון כל ההפניות בפרויקט, לחץ Yes.

2. באמצעות חלון Properties שנה את המאפיין Text של הטופס ל- **Products Maintenance**.

3. הוסף לטופס פקד **Label**. שנה את המאפיין **Text** של הפקד ל- **Number of Products** ואת המאפיין **Location** שנה ל- **"25, 34"** (ללא גרשיים).

4. הוסף לטופס פקד **Label**. שנה את המאפיין Location של הפקד החדש ל- **"131, 34"** (ללא גרשיים), ואת המאפיין (**Name**) והמאפיין **Text** שנה ל-**numProducts**.

5. הרחב את המאפיין (**DataBindings**) של הפקד **numProducts**. לחץ על המאפיין **Text** (נמצא בתוך **DataBindings**), ואחר כך לחץ על התפריט הנפתח שיופיע.

כעת יופיע חלון שמציג עץ של מקורות נתונים. הרחב את הרכיבים Other Data Sources, Project Data Sources, NorthwindDataSet, NumProductsTable ובחר NumProducts. פעולה זו כורכת את המאפיין **Text** של התווית **Label** לעמודה **NumProducts** בטבלת הנתונים **NumProductsTable**.

סביבת הפיתוח תחולל מופע **northwindDataSet** של המחלקה **NorthwindDataSet**, מופע **numProductsTableAdapter** של המחלקה **NumProductsTableAdapter**, ואובייקט **numProductsTableBindingSource** מסוג **BindingSource**. אחר כך הוא מצרף אותם לטופס.

.6 לחץ על האובייקט **numProductsTableBindingSource** שתחת הטופס. בחלון Properties בחר במאפיין **DataSource**, אשר מזהה את האובייקט **DataSet** שמשמש את **BindingSource** לביצוע ההתחברות לבסיס הנתונים. שים לב שהוא מפנה לאובייקט **northwindDataSet**.

.7 לחץ על המאפיין **DataMember**. מאפיין זה מצביע על האובייקט **DataTable** שב-**northwindDataSet**, אשר משמש כמקור הנתונים. נכון לעכשיו, אובייקט זה הוא **NumProductsTable**.

.8 לחץ שוב על הפקד **numProducts**, והרחב את המאפיין (**DataBindings**). בדוק שוב את המאפיין **Text**. שים לב שכעת הוא מקובע על **NumProductsTableBindingSource - NumProducts**.

.9 הצג את הקובץ ProductsForm.cs בחלון Code and Text Editor ומצא את השיטה ProductsForm_Load.

שיטה זו, שנוצרה על ידי סביבת העבודה, מופעלת כאשר הטופס נפתח. היא מכילה את המשפט הבא:

```
this.numProductsTableAdapter.Fill(this.northwindDataSet.
NumProductsTable);
```

על בסיס קבוצה זו של אובייקטים, מאפיינים והקוד עצמו, אתה צריך להבין כיצד הפקד numProducts מאחזר את הנתונים מתוך בסיס הנתונים:

א. האובייקט **numProductsTableAdapter** מתחבר לבסיס הנתונים ומאחזר על ידי הפעלת משפט SQL SELECT את השורות מהטבלה Products שבבסיס הנתונים.

ב. האובייקט **numProductsTableAdapter** משתמש במידע זה כדי למלא בנתונים את טבלת הנתונים **NumProductsTable** שבאובייקט **northwindDataSet**, כאשר הטופס נפתח.

ג. האובייקט **numProductsTableBindingSource** מחבר בין העמודה **NumProducts** שבטבלת הנתונים **NumProductsTable** השייכת לאובייקט **northwindDataSet**, לבין המאפיין **Text** של הפקד numProducts.

10. בנה והפעל את היישום.

הפקד **Label** יציג את מספר השורות בטבלה Products (77, אלא אם כן שינית את הנתונים בבסיס הנתונים מאז שהוא נוצר).

11. סגור את הטופס וחזור לסביבת הפיתוח של Visual Studio 2005.

כריכת נתונים דינמית

באפשרותך לקבוע את המאפיין **DataBindings** של הפקד גם על ידי כתיבת קוד המופעל בזמן ההרצה וגם באופן סטטי בזמן העיצוב. המאפיין **DataBindings** מכיל את השיטה **Add** שניתן להשתמש בה בכדי לכרוך מאפיין אל מקור מידע. לדוגמה, ניתן לכרוך את המאפיין **myLabel** מסוג **Text** של פקד **Label** לעמודה **NumProducts** שבאובייקט **numProductsTableBindingSource**. הנה כך:

```
myLabel.DataBindings.Add("Text", numProductsTableBindingSource,
  "NumProducts");
```

הפרמטרים של השיטה **Add** הם שם המאפיין שיש לכרוך, שם האובייקט המכיל את הנתונים שיש לכרוך, וחבר המחלקה אשר מספק למעשה את הנתונים.

ניתן להשתמש בשיטה **Add** כדי לכרוך למאפיינים של פקדים אחרים. הדוגמה הבאה כורכת את המאפיין **Text** של **myLabel** אל המאפיין **Text** של פקד **myText** מסוג **TextBox**:

```
myLabel.DataBindings.Add("Text", myText, "Text");
```

כשתריץ את הקוד, תראה שהתווית מתעדכנת באופן אוטומטי כאשר אתה מקליד משהו בתיבת הטקסט.

כריכת נתונים מורכבת

בסעיפים קודמים למדת להשתמש בכריכת נתונים פשוטה, כדי לשייך מאפיין של פקד אל ערך בודד במקור הנתונים. כריכת נתונים מורכבת מיועדת למקרה שברצונך להציג רשימה של ערכים ממקור הנתונים. בתרגילים הבאים תצטרך להשתמש בכריכת נתונים מורכבת כדי להציג בפקד **ComboBox** את שמות הספקים שמופיעים בבסיס הנתונים, ולאחר מכן להציג את קוד הספק (SupplierID) הנבחר על ידי המשתמש. בהמשך הפרק, כאשר נעסוק בשיפור היישום Products Maintenance, תתרגל את הטכניקות שתלמד כעת.

טיפ

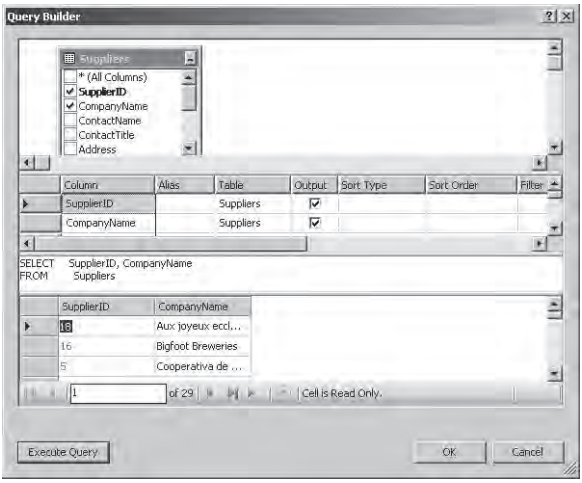
גם הפקדים **ListBox** ו-**CheckedListBox** תומכים בכריכת נתונים מורכבת שמבוצעת בטכניקה זהה לזו שמוצגת להלן.

צור מקור נתונים וקבע את תצורתו

1. ב- Solution Explorer, לחץ לחיצה כפולה על NorthwindDataSet.xsd כדי להציג את החלון DataSet Designer.
2. השתמש בערכת הכלים כדי להוסיף **TableAdapter** לאובייקט **DataSet**.
   כעת יופעל האשף TableAdapter Configuration.
3. באשף TableAdapter Configuration בחר להתחבר לבסיס הנתונים באמצעות NorthwindConnectionString ולחץ Next.
4. בעמוד Choose a Command Type, בחר Use SQL Statements ולחץ Next.
5. בעמוד Enter a SQL Statement, לחץ Query Builder.
   באמצעות תיבת הדו-שיח Query Builder מספק האשף דרך נוחה להרכבת משפט SELECT במידה ואינך זוכר את התחביר של SQL.
6. בתיבת הדו-שיח Add, בחר בטבלה Suppliers, לחץ Add ואז לחץ Close.
   הטבלה Suppliers תתווסף לתיבת הדו-שיח Query Builder.
7. סמן את העמודות SupplierID ו-CompanyName. שים לב שהחלונית התחתונה בתיבת הדו-שיח מציגה את משפטי SQL SELECT השייכים לעמודות שאתה בוחר:

```
SELECT SupplierID, CompanyName FROM Suppliers
```

כדי לוודא שהמשפטים אכן מאחזרים את השורות הנכונות, לחץ על הלחצן Execute Query. תוצאות השאילתה יוצגו בחלונית שבתחתית תיבת הדו-שיח.



8. לחץ OK כדי לסגור את Query Builder ולחזור לאשף TableAdapter Configuration.
   המשפט SELECT יועתק לדף Enter a SQL Statement.

9. יישום זה אינו משמש לעריכת פרטי הספקים, ולכן לחץ Advanced Options ובטל את תיבת הסימון Generate Insert, Update and Delete Statements. לחץ OK ואחר כך לחץ Finish.

מחלקה **Suppliers** חדשה מסוג **DataTable** ומחלקה **SuppliersTableAdapter** חדשה מסוג **DataTableAdapter** יתווספו לחלון DataSet Designer.

10. בנה מחדש את הפתרון בכדי לחולל את הקוד עבור האובייקטים החדשים.

כריכת תיבה משולבת לטבלת הנתונים

1. הצג את הטופס ProductsForm בחלון תצוגת העיצוב, כדי לחולל בו **תיבה משולבת** (**combo box**) ופקד תווית (label).

2. באמצעות ערכת הכלים, הוסף לטופס פקד **ComboBox** ופקד **Label**. שנה את ערכי המאפיינים של הפקדים על פי הערכים שבטבלה הבאה:

Control	Property	Value
comboBox1	(Name)	supplierList
	Location	25, 65
label2	(Name)	supplierID
	Location	178, 70
	Text	SupplierID

3. לחץ על הפקד **ComboBox** שבטופס. בחלון Properties בחר במאפיין **DataSource**. לחץ על התפריט הנפתח שיופיע. הרחב את Other Data Sources, Project Data Sources, Northwind DataSet, ואז לחץ על Suppliers.

4. את המאפיין **DisplayMember** של התיבה המשולבת שנה ל-**CompanyName**, ואת המאפיין **ValueMember** שנה ל-**SupplierID**.

כאשר הטופס פועל, הוא יציג את רשימת הספקים בתיבה המשולבת. כאשר המשתמש בוחר באחד הספקים, קוד הספק (SupplierID) יופיע בתור ערך של התיבה המשולבת.

5. בחלון Properties לחץ Events כאשר התיבה המשולבת מסומנת, ולחץ לחיצה כפולה על האירוע **SelectedIndexChanged** של התיבה המשולבת.

הקוד של ProductForm יוצג כעת בחלון Code and Text Editor, ובנוסף ייווצר המטפל באירוע **supplierList_SelectedIndexChanged**.

6. הוסף לשיטה **supplierList_SelectedIndexChanged** את המשפטים האלה:

```
if (supplierList.SelectedValue != null)
{
   supplierID.Text = supplierList.SelectedValue.ToString();
}
```

בלוק זה של קוד מציג בתווית **supplierID** את קוד הספק אותו בחר המשתמש. המאפיין **SelectedValue** מחזיר את ערך העמודה **ValueMember** שבשורה הנוכחית. הערך מוחזר כאובייקט, ולכן עליך להשתמש בשיטה **ToString** אם אתה רוצה לעבד אותו כמחרוזת.

7. עיין בקוד של השיטה **ProductsForm_Load**. שים לב לצירוף שורת קוד אשר מציינת שהשיטה **Fill** של **suppliersTableAdapter** ממלאת את טבלת הנתונים **Suppliers** אשר באובייקט **northwindDataSet**.
8. בנה והפעל את היישום. התיבה המשולבת מציגה את שמות כל הספקים. בחר אחד מהם. קוד הספק יופיע בתווית שמימין לתיבה המשולבת. בחר בשם של ספק אחר וודא שהקוד שלו אכן מעודכן בתווית.
9. סגור את הטופס וחזור ל- Visual Studio 2005.

עדכון בסיס הנתונים באמצעות DataSet

בתרגילים הקודמים בפרק זה למדת לאחזר מבסיס הנתונים אל אובייקטים בזיכרון. כעת הגיע הזמן שתלמד לעדכן נתונים. תחילה עליך לסקור את הבעיות הפוטנציאליות וללמוד להתגבר עליהן באמצעות **DataSet**.

בסיסי הנתונים צריכים לתמוך בהתחברות של משתמשים אחדים במקביל, אולם משאבי החיבור במקביל עשויים להיות מוגבלים. ביישומים המשמשים לאחזור ולהצגה של נתונים, לעולם אינך יכול להיות בטוח כמה זמן המשתמש ישוטט בבסיס הנתונים, למרות שאין זה מומלץ לפתוח חיבור לבסיס נתונים לפרקי זמן ארוכים. גישה טובה יותר תהיה להתחבר לבסיס הנתונים, לאחזר את הנתונים לאובייקט **DataSet** לשם עיבוד, ואז לסגור את החיבור. כך המשתמש יוכל לשוטט בין הנתונים השונים שבאובייקט **DataSet** ולבצע את השינויים הרצויים. כאשר המשתמש רוצה לשמור את השינויים שביצע, היישום יכול לפתוח שוב את החיבור לבסיס הנתונים ולשלוח את השינויים. תהליך זה יוצר מספר קשיים שצריך ללמוד להתגבר עליהם, כמו למשל, מה קורה אם שני משתמשים מבצעים עדכונים שונים באותם נתונים. איזה מבין שני הערכים צריך היישום לשמור בבסיס הנתונים? נחזור לבעיה זו בהמשך.

ניהול החיבורים

בתרגילים הקודמים ראית שבעת הגדרת **DataSet** עליך לציין את החיבור אשר ישמש להתקשרות עם בסיס הנתונים. מידע זה נקבע במתאם **TableAdapter** המשמש לאחזור הנתונים ולהצבתם באובייקט **DataSet**. בעת הפעלת השיטות **Fill** או **GetData**, הקוד שחוללה סביבת הפיתוח בודק תחילה את מצב החיבור. אם החיבור כבר פתוח, הוא ישמש לאחזור הנתונים, ויישאר פתוח גם בסיום הפעולה. אם החיבור סגור, השיטות **Fill** ו-**GetData** פותחות אותו, מאחזרות את הנתונים ולבסוף סוגרות אותו. במקרה כזה, האובייקט **DataSet** הינו **מנותק (disconnected)**, כי אינו מחזיק באופן קבוע חיבור פעיל אל בסיס הנתונים. אובייקטים מסוג DataSet מנותק משמשים ביישומים כמו data cache. את הנתונים שבאובייקט DataSet ניתן לשנות, ובשלב מאוחר יותר לפתוח שוב את החיבור ולשלוח דרכו את השינויים אל בסיס הנתונים.

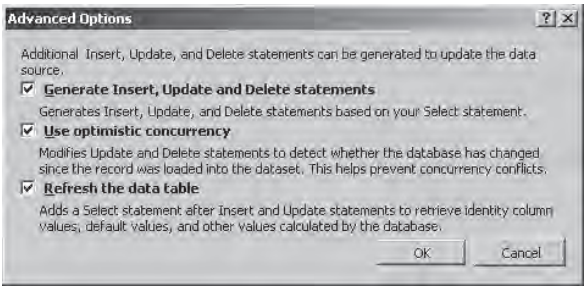
כדי לפתוח באופן ידני את החיבור אל בסיס הנתונים, עליך ליצור אובייקט **SqlConnection** ולקבוע את המאפיין **ConnectionString** שלו. לבסוף עליך לקרוא לשיטה **Open** כפי שמוסבר בפרק 23. ניתן לשייך חיבור פתוח למתאם **TableAdapter** על ידי שימוש במאפיין **Connection** שלו. הקוד הבא ממחיש כיצד להתחבר לבסיס הנתונים ולמלא את טבלת הנתונים **Suppliers**. בדוגמה זו החיבור ל**בסיס הנתונים** יישאר פתוח גם לאחר סיום השיטה Fill:

```
SqlConnection dataConnection = new SqlConnection();
dataConnection.ConnectionString = “Integrated Security=true;” +
               “Initial Catalog=Northwind;” +
               “Data Source=YourServer\\
                SQLExpress”;
dataConnection.Open();
suppliersTableAdapter.Connection = dataConnection;
suppliersTableAdapter.Fill(northwindDataSet.Suppliers);
```

רק אם יש לך סיבה ממש טובה לעשות זאת, הימנע מהשארת חיבורים פתוחים לפרקי זמן ארוכים מהנחוץ. עדיף ליצור **DataSet** מנותק ולאפשר לשיטות **Fill** ו-**GetData** לפתוח ולסגור עבורך את החיבורים.

טיפול בעדכונים שנעשים במקביל

כבר הצגנו את הבעיה המתעוררת כאשר שני משתמשים מנסים לעדכן את אותם הנתונים באותו זמן. יש לפחות שתי גישות שמאפשרות לפתור בעיה זו. לכל אחת מהגישות יש יתרונות וחסרונות שונים. הטכניקה הראשונה עושה שימוש באפשרות Use optimistic concurrency שבתיבת הדו-שיח Advanced Options שבאשף TableAdapter Configuration.



אם תבטל את תיבת הסימון שלצד אפשרות זו, השורות שמועברות לאובייקט **DataSet** תינעלנה בבסיס הנתונים בכדי למנוע ממשתמשים אחרים לשנותן. גישה זו מכונה pessimistic concurrency, והיא מבטיחה על ידי חסימת משתמשים אלה, שהשינויים שמבוצעים על ידי המשתמש לא יתנגשו עם שינויים שהם רוצים לערוך. אבל, כאשר אתה מאחזר מספר רב של שורות אך משנה רק חלק קטן מהן, אתה עשוי לחסום את המשתמשים האחרים מביצוע שינויים בכל השורות שאחזרת. ישנו חיסרון נוסף - נעילת המידע מחייבת להשאיר את החיבור לבסיס הנתונים פתוח, ולכן השימוש בגישה pessimistic concurrency עשויה גם לגרום לצריכה גבוהה של משאבי התחברות. היתרון המרכזי של גישה זו הוא כמובן הפשטות, מכיוון שאינך צריך לכתוב קוד שמטרתו לבדוק שינויים שנעשו על ידי משתמשים אחרים לפני עדכון בסיס הנתונים על ידך.

האפשרות "Use optimistic concurrency" מונעת נעילת נתונים, ולכן גם ניתן לסגור את החיבור לאחר האחזור מבסיס הנתונים. החיסרון הוא בכך שעליך לכתוב קוד שנועד לגלות אם המשתמש ביצע עדכונים אשר סותרים עדכונים שביצעו משתמשים אחרים. סוג כזה של קוד קשה לכתיבה וקשה גם להדר אותו. אולם, האובייקט **TableAdapter** שמחולל האשף TableAdapter Configuration מסתיר חלק גדול מהמורכבות, למרות שעליך בכל זאת להיות מוכן לטפל באירועים שעלולים לצוף במקרה של סתירה. נרחיב את הדיון בנושא זה בחלק האחרון של פרק זה.

שימוש באובייקט DataSet בעזרת פקד DataGridView

לאחר שלמדת ליצור אובייקט מסוג **DataSet**, לאחזר שורות ולהציג נתונים, עליך להוסיף ליישום Products Maintenance אלמנט המאפשר למשתמש לעדכן את פרטי המוצר בבסיס הנתונים. עליך להציג ולעדכן את המידע באמצעות הפקד **DataGridView**.

הוספת הפקד DataGridView לטופס

1. הצג את הטופס ProductsForm בחלון תצוגת העיצוב.
2. בחלון Code and Text Editor הסר את השיטה **supplierList_SelectedIndexChanger** מהקובץ ProductsForm.cs.
3. חזור לחלון תצוגת העיצוב. החלף את המאפיין Size של הטופס ל- **"600, 400"** כדי לשנות את גודלו.
4. בערכת הכלים הרחב את הקטגוריה Data ולחץ על הפקד **DataGridView**. הצב את הפקד על הטופס. החלף את שם הפקד **DataGridView** ל-**productsGrid**. את המאפיין **Location** שלו החלף ל- **"13, 61"** ואת המאפיין Size החלף ל- **"567, 300"**.
5. הוסף שני פקדי **Button** לטופס, הצב אותם מעל **productsGrid** והחלף את מאפייני **Location** שלהם ל- **"402, 22"** ו- **"505, 22"**. החלף את שמותיהם ל-**queryButton** ו-**saveButton**. החלף את המאפיין **Text** של כל אחד מהמאפיינים ל-**Query** ו-**Save** בהתאמה.

השלב הבא הוא ליצור מחלקה **DataAdapter** ולכרוך אותה לפקד **DataGridView**. עליך להיעזר באשף TableAdapter Configuration כדי ליצור את המחלקה DataAdapter. בשביל הגיוון, במקום לכרוך את הפקד **DataGridView** על ידי קיבוע המאפיינים **DataSource** ו-**DataMember** שלו באמצעות חלון תצוגת העיצוב, עליך לכרוך את המתאם **DataAdapter** על ידי כתיבת קוד מתאים.

יצירת DataAdapter לאחזור מוצרים וכריכתו לפקד DataGridView

1. ב- Solution Explore, לחץ לחיצה כפולה על הקובץ NorthwindDataSet.xsd כדי להציג את החלון DataSet Design.

2. היעזר באשף TableAdapter Configuration כדי להוסיף מתאם **TableAdapter** חדש לאובייקט **DataSet**, שתפקידו אחזור הנתונים של המוצרים מבסיס הנתונים Northwind. תוכל להיעזר בטבלה הבאה.

הדף באשף	שדה	ערך
Choose Your Data Connection	באיזה נתוני התחברות היישום צריך להשתמש כדי להתחבר לבסיס הנתונים?	**NorthwindConnectionString**
Choose a Command Type	באיזו דרך המתאם **TableAdapter** צריך להתחבר לבסיס הנתונים?	**Use SQL Statement**
Enter a SQL Statement	אילו נתונים יש להעביר לטבלה?	**SELECT * FROM Products**
	אפשרויות מתקדמות Advanced Options	**סמן את כל האפשרויות**
Choose Methods to Generate	אילו שיטות יש להוסיף לאובייקט **TableAdapter**	**בחר בכל האפשרויות, ובחר בשמות ברירת המחדל של השיטות.**

שים לב שבסיום עבודתך עם האשף, ולאחר יצירת האובייקטים **DataTable** ו-**TableAdapter**, האשף מזהה באופן אוטומטי שהטבלאות Suppliers ו-Products קשורות זו לזו, ולכן יוצר את **Relation** אשר מקשר בין שתי טבלאות הנתונים.

3. בתפריט Build בחר Rebuild Solution כדי לחולל את הקוד עבור המחלקות **Data Table** ו-**TableAdapter** החדשות.

4. הצג את הטופס ProductionForm בחלון תצוגת העיצוב. לחץ לחיצה כפולה על פקד הלחצן **Query**.

כתוצאה, סביבת הפיתוח מחוללת עבור הפקד מטפל באירוע (event handler) בשם **queryButton_Click**, ומעבירה אותך לחלון Code and Text Editor.

5. הוסף את המשפטים הבאים לשיטה **queryButton_Click**:

```
NorthwindDataSetTableAdapters.ProductsTableAdapter productsTA =
        new NorthwindDataSetTableAdapters.
ProductsTableAdapter();
productsTA.Fill(northwindDataSet.Products);
BindingSource productsBS =
new BindingSource(northwindDataSet, “Products”);
productsGrid.DataSource = productsBS;
```

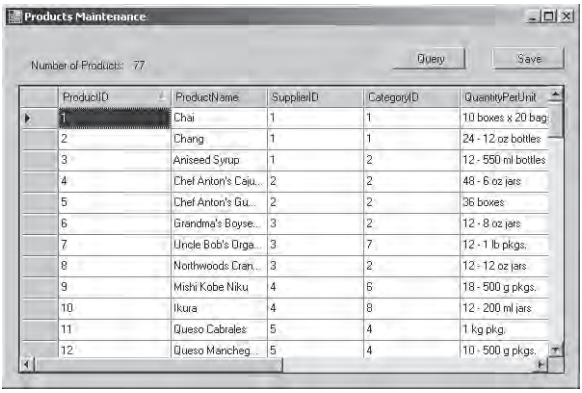
המשפט הראשון יוצר מופע חדש של המחלקה **ProductsTableAdapter** שהגדרת על ידי האשף TableAdapter Configuration. שים לב שמחלקה זו, בדומה לשאר מחלקות **TableAdapter** של האובייקט **DataSet** הנוכחי, מוגדרת במסגרת מרחב השמות (namespace) שנקרא **NortwindDataSetTableAdapter**. המשפט השני משתמש באובייקט זה כדי למלא את טבלת הנתונים Products שבאובייקט **NortwindDataSet**.

עליך לזכור שמשפט זה יגרום לניתוק אוטומטי מבסיס הנתונים לאחר אחזור הנתונים, מכיוון שלא נמצא חיבור פתוח לפני האחזור. המשפט השלישי יוצר אובייקט **BindingSource** חדש עבור טבלת הנתונים Products שבאובייקט **NortwindDataSet**. המשפט הרביעי הוא זה המבצע למעשה את כריכת הנתונים - הוא קובע את ערכי המאפיין **DataSource** של הפקד **DataGridView** כדי שיפנה לאובייקט **BindingSource** החדש.

.6 בנה והפעל את היישום.

בעת הפעלת היישום, הפקד **DataGridView** עודנו ריק.

.7 לחץ **Query**. הפקד **DataGridView** יציג את רשימת המוצרים. ודא שמספר השורות בפקד DataGridView שווה לערך המוצג על ידי התווית Number of Products.



.8 לחץ על כותרת העמודה **ProductName**. השורות יסתדרו בסדר עולה לפי שם המוצר. לחץ שוב על כותרת העמודה. הפעם השורות יסתדרו בסדר יורד. לחץ על כותרת העמודה **ProductID** כדי להחזיר את השורות לסדרן המקורי.

.9 לחץ על אחד מהתאים שבפקד **DataGridView** ושנה את הנתון המאוחסן בו. על פי ברירת המחדל ניתן לשנות את תוכן כל התאים מלבד ProductID. אינך יכול לשנות את קוד המוצר, מכיוון שעמודה זו משמשת כ- primary key של הטבלה בבסיס הנתונים. זכור שאסור לשנות את ערכי המפתח הראשי, כי הם משמשים לזיהוי השורה בבסיס הנתונים.

.10 גלול לימין כדי להציג את העמודה **Discontinued**. שים לב שעמודה זו מופיעה בתור תיבת סימון. בבסיס הנתונים עמודה זו הינה מסוג **bit** שמכיל שני ערכים בלבד (1 או 0).

.11 גלול מטה לתחתית הטבלה שבפקד **DataGridView**, שבה יש שורה שמסומנת בכוכבית ומשמשת להוספת מוצר חדש. שים לב ש- ProductID נוצר באופן אוטומטי.

.12 לחץ על המרווח האפור שבצד השמאלי של שורה 76, ואז כל השורה מודגשת. לחץ Delete והשורה תיעלם מתצוגת הפקד DataGridView.

.13 לאחר שתסיים לעיין בנתונים, סגור את הטופס וחזור אל סביבת הפיתוח. השינויים שביצעת לא יישמרו, מכיוון שטרם כתבת את הקוד לביצוע השמירה.

בדיקת תקינות לקלט משתמש באמצעות הפקד DataGridView

לפני שמירת השינויים בבסיס הנתונים, עליך לוודא שהשינויים שבוצעו על ידי המשתמש אמנם תקינים. במצב הנוכחי אין הגבלה על תוכן הנתונים שהמשתמש מזין אל הפקד **DataGridView**. בתרגילים הבאים, תלמד כיצד להגביל את הקלט מהמשתמש לערכים או לסוגים מסוימים, בכדי למנוע שגיאות בנתונים שבפקד, ולקבוע שהם תקינים.

קביעת תצורת הפקד DataGridView להגבלת הקלט מהמשתמש

1. הצג את הטופס ProductsForm בחלון תצוגת העיצוב. מחק את הלחצן Query.
2. הצג את הקובץ ProductsForm.cs בחלון Code and Text Editor. הפוך את השיטה **queryButton_Click** ואת התוכן שלה להערה.
3. חזור לחלון תצוגת העיצוב ולחץ על הפקד **DataGridView**. בחלון Properties, קבע את המאפיין **DataSource** לטבלת הנתונים Products שבמחלקה **NorthwindDataSet** (שב-**Project Data Sources**, שב-Other Data Sources).

   עמודות הטבלה Products יופיעו בפקד DataGridView.
4. בחלון Properties בחר במאפיין **Columns**, ולחץ על לחצן ההמשך (ellipses button).

   תיבת הדו-שיח Edit Columns תוצג ותוכל לצפות בה במאפיינים של כל אחת מהעמודות שבפקד **DataGridView** וגם לערוך אותם.
5. בתיבת הדו-שיח Edit Columns, לחץ על העמודה ProductID. ברשימה Bound Column Properties, ודא שערך המאפיין **ReadOnly** הוא True. הערכים שבעמודה זו נוצרים באופן אוטומטי. בכך שאנו מונעים מהמשתמש לשנות אותם, אנו שוללים למעשה את אחד המקורות האפשריים לשגיאה.
6. לחץ על העמודה SupplierID. שנה את המאפיין **ColumnType** ל-**DataGridViewComboBoxColumn** ואת המאפיין DisplayStyle שנה ל-ComboBox. כעת עמודה זו תוצג כתיבה משולבת ולא כתיבת טקסט.

הערה

המאפיין **DisplayStyle** יופיע רק לאחר שתשנה את ערך המאפיין **ColumnType**.

7. שנה את המאפיין **DataSource** ל-**suppliersBindingSource**. עליך להציג בעמודה זו רשימה של ספקים על ידי שימוש בטכניקת כריכת הנתונים המורכבת שלמדת. שנה את המאפיין **DisplayMember** ל-**CompanyName**, ואת המאפיין **ValueMember** שנה ל-**SupplierID**. כאשר המשתמש בוחר באחד הספקים מהרשימה, מנגנון הפעולה הפנימי של היישום ישתמש בקוד הספק (SupplierID) כדי למצוא את העמודה בטבלה Products. כך אנחנו שוללים עוד מקור אפשרי לשגיאה.
8. בתיבת הדו-שיח Edit Columns לחץ OK.

למדת כיצד למנוע מספר שגיאות בסיסיות על ידי הגבלת הקלט המתקבל מהמשתמש לקבוצת ערכים תקניים שנקבעים מראש. אולם, צריך ללכוד שגיאות אחרות שעשויות לקרות. לצורך כך, עליך לכתוב מטפלים (handlers) עבור האירועים **CellValidating**, **CellEndEdit** ו-**DataError**. האירוע **CellValidating** מתרחש כל פעם שהמשתמש משנה תוכן אחד התאים. האירוע CellEndEdit מופעל לאחר שתא עודכן והמשתמש מנסה לעבור לתא הבא.

טיפול באירועים CellEndEdit, CellValidating ו-DataError

1. בחלון תצוגת העיצוב, לחץ על הפקד **productsGrid**, ובחלון Properties לחץ Events.
2. לחץ לחיצה כפולה על האירוע **CellValidating**.

   Visual Studio 2005 תיצור עבורך את שיטת האירוע **productsGrid_ CellValidating**.
3. הוסף את המשפטים הבאים לשיטה **productsGrid_CellValidating**:

```
int newInteger;
productsGrid.Rows[e.RowIndex].ErrorText = “”;
if ((productsGrid.Columns[e.ColumnIndex].DataPropertyName ==
     “UnitsInStock”) ||
    (productsGrid.Columns[e.ColumnIndex].DataPropertyName ==
     “UnitsOnOrder”) ||
    (productsGrid.Columns[e.ColumnIndex].DataPropertyName ==
     “ReorderLevel”))
{
    if (!int.TryParse(e.FormattedValue.ToString(), out
        newInteger) || newInteger < 0)
    {
        productsGrid.Rows[e.RowIndex].ErrorText =
           “Value must be a non-negative number”;
        e.Cancel = true;
    }
}
```

שיטה זו מוודאת שהמשתמש הזין מספר שלם שאינו שלילי בעמודות UnitsInStock, UnitsOnOrder ו-ReorderLevel.

הפרמטר השני של השיטה, **e**, הוא אובייקט **DataGridViewCellValidatingEventArgs**. הוא כולל מספר מאפיינים אשר יכולים לשמש אותך כדי לאתר את התא שערכת. הפרמטר **e.ColumnIndex** מכיל את מספר העמודה, ו-**e.RowIndex** מכיל את מספר השורה (זכור שהעמודה הראשונה היא 0, והשורה הראשונה היא 0). משפט **if** הראשון קובע באיזו עמודה המשתמש נמצא. שים לב שהמחלקה **DataGridView** מכילה אוסף של פריטים מסוג **Column** ובו מאוחסן המידע על כל אחת מהעמודות המוצגות. הערך של **e.ColumnIndex** משמש כאינדקס של האוסף הזה, וערכו של המאפיין **Name** משמש לזיהוי העמודה.

השיטה **int.TryParse** מאפשרת לדעת אם מחרוזת כלשהי מכילה ערך שניתן להמיר למספר שלם. היא מחזירה ערך **true** אם ההמרה מוצלחת (ומעבירה **int** שמכיל את הערך המומר כפרמטר **out**); ומחזירה ערך **false** ההמרה נכשלה. משפט **if** השני מפעיל את השיטה **int.TryParse** כדי לקבוע אם המשתמש הקליד ערך שלם תקני בתא הנוכחי.

אם הבדיקה נכשלת, או אם הערך תקני אך קטן מאפס - תוצב במאפיין **ErrorText** של השורה הנוכחית הודעת השגיאה המתאימה. המאפיין **ErrorText** מתנהג בדומה לפקד **ErrorProvider** שהכרת בפרק 22 - הוא מציג סמל שגיאה בעת הצורך, והמשתמש יכול להציב את סמן העכבר מעל הסמל כדי לקרוא את הודעת השגיאה. המאפיין **Cancel** של הפרמטר **e** נקבע כדי למנוע מהמשתמש לעבור לתא אחר כל עוד לא הזין בו נתונים תקפים, שאינם שגויים.

.4 בחלון תצוגת העיצוב, לחץ על הפקד **DataGridView**. בחלון Properties לחץ Events. לחץ לחיצה כפולה על האירוע **CellEndEdit**.

Visual Studio 2005 תיצור עבורך מטפל באירוע **CellEndEdit** בשם **productsGrid_CellEndEdit**, ותעביר אותך לחלון Code and Text Editor.

.5 הוסף את המשפט הבא לשיטה **productsGrid_CellEndEdit**:

```
productsGrid.Rows[e.RowIndex].ErrorText = “”;
```

שיטה זו מופעלת לאחר בדיקת תקינות הנתונים וכאשר המשתמש רוצה לעבור לתא הבא. מטרת הקוד למחוק את הודעות השגיאה המוצגות.

.6 בחלון תצוגת העיצוב, לחץ על הפקד **DataGridView**. בחלון Properties לחץ Events. לחץ לחיצה כפולה על האירוע **DataError**.

Visual Studio 2005 תיצור עבורך מטפל באירוע **DataError** בשם **productsGrid_DataError**, ותעביר אותך לחלון Code and Text Editor.

.7 הוסף את המשפט הבא לשיטה **productsGrid_DataError**:

```
productsGrid.Rows[e.RowIndex].ErrorText = “Invalid input.
Please re-enter”; e.Cancel = true;
```

האירוע **DataError** נועד ללכוד את כל החריגים שנוצרו עקב כישלון בבדיקת תקינות הנתונים. כאשר המשתמש מקליד נתון שאינו תקני (כגון מחרוזת אלפביתית בעמודה נומרית), וערך זה לא נלכד על ידי אחד האירועים האחרים, הוא יילכד על ידי אירוע זה. כדי לממש את המטפל באירוע, עליך להציב הודעת שגיאה במאפיין **ErrorText** ולמנוע מהמשתמש לעבור לפעולה הבאה.

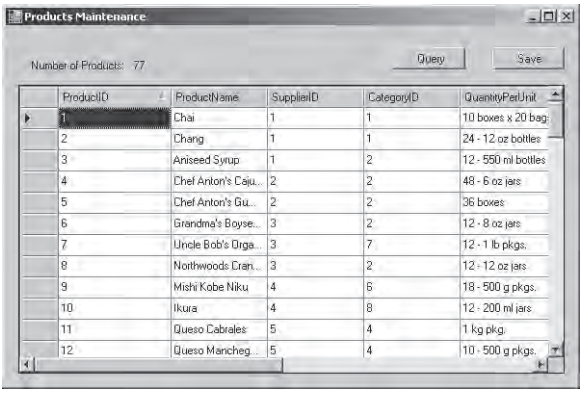
.8 בנה והפעל את היישום.

הפקד **DataGridView** מתמלא בנתונים עם הופעת הטופס.

.9 נסה לשנות את הערך בעמודה ProductID. עמודה זו אמורה להיות לקריאה בלבד, ולכן לא תצליח לעשות זאת.

.10 שנה את ה- SupplierID של אחת השורות באמצעות התפריט הנפתח אשר מציג את שמות הספקים השונים.

.11 הקלד מספר שלם שלילי בעמודה UnitsInStock ונסה לעבור אל אחד התאים האחרים. סמל שגיאה יופיע בכותרת השורה. הצב את סמן העכבר מעל הסמל כדי להציג את הודעת השגיאה שמתקבלת מהאירוע **CellValidating**.



12. הקלד ערך שלם חיובי בעמודה UnitsInStock ועבור לתא אחר. סמל השגיאה ייעלם והמידע יתקבל.

13. הקלד מחרוזות אלפביתיות בעמודה CategoryID ועבור לתא אחר. סמל השגיאה יופיע שוב. הצב את סמן העכבר מעל הסמל, כדי להציג את הודעת השגיאה, שמתקבלת מהאירוע **DataError**.

14. הקש **Escape** כדי לבטל את השינוי.
סמל השגיאה ייעלם.

15. סגור את הטופס וחזור אל Visual Studio 2005.

עדכונים באמצעות DataSet

השינויים המבוצעים באמצעות הפקד **DataGridView** מועתקים באופן אוטומטי אל אובייקט **DataSet** אשר מהווה את מקור הנתונים של הפקד. כדי לשמור את השינויים, צריך לפתוח שוב את החיבור לבסיס הנתונים, להפעיל את המשפטים SQL INSERT, UPDATE ו-DELETE הדרושים ולבסוף - לסגור את החיבור אל בסיס הנתונים. עליך גם להיערך לטיפול בשגיאות אפשריות. למרבה המזל, המחלקה **TableAdapter** שנוצרה עבור האובייקט **DataSet** כוללת מספר שיטות ואירועים שיכולים לסייע בביצוע המשימות הללו.

לפני עדכון בסיס הנתונים, עליך לוודא את תקינות הנתונים. אחרי הכל, אם תתחבר ותתנתק מבסיס הנתונים כדי לעדכן נתונים שבסופו של דבר יתגלו כמוטעים, יהיה זה בזבוז משאבים.

ודא את תקינות העדכונים וטפל בשגיאות

1. הצג את הטופס ProductsForm בחלון תצוגת העיצוב. בחר בלחצן Save. בחלון Properties לחץ Events. לחץ לחיצה כפולה על האירוע Click.
סביבת הפיתוח תחולל שיטת אירוע בשם **saveButton_Click**.

2. בחלון Code and Text Editor הוסף לשיטה **saveButton_Click** את הבלוק **try/catch** הבא:

```
try
{
    NorthwindDataSet changes = (NorthwindDataSet)northwindData
    Set.GetChanges();
    if (changes == null)
    {
        return;
    }
    // Check for errors
    // If no errors then update the database, otherwise tell
       the user
}
catch (Exception ex)
{
    MessageBox.Show("Error: " + ex.Message, "Errors",
       MessageBoxButtons.OK, MessageBoxIcon.Error);
    northwindDataSet.RejectChanges();
}
```

בלוק השיטה **GetChanges** של **northwindDataSet** משמש ליצירת אובייקט חדש מסוג **NorthwindDataSet** אשר מכיל רק את השורות אשר שונו. שים לב שההשלכה (cast) שבקוד הכרחית, כי השיטה **GetChanges** מחזירה אובייקט **DataSet** גנרי.

למרות שקטע זה של קוד אינו מחייב, הוא מקל מאוד על עדכון בסיס הנתונים, מכיוון שכך תהליכי העדכון אינם צריכים לבדוק אילו שורות שונו ואילו שורות לא שונו. אם לא נמצאו שינויים (ערך האובייקט **NorthwindDataSet** הוא null) השיטה תסתיים; ואם כן יימצאו שינויים, השיטה תחפש שגיאות בנתונים ולאחר מכן תעדכן את בסיס הנתונים (בהמשך תכתוב את הקוד שעושה זאת). אם נזרק חריג בזמן העדכון, היישום יציג הודעה בפני המשתמש ויבטל את השינויים שבוצעו באובייקט **northwindDataEditor** באמצעות השיטה **RejectChanges**.

3. בחלון Code and Text Editor, החלף את ההערה // Check for errors שבשיטה **saveButton_Click** בקטע הקוד הבא:

```
DataTable dt = changes.Tables["Products"];
DataRow [] badRows = dt.GetErrors();
```

המשפט הראשון מאחזר את טבלת הנתונים Products שבאובייקט **DataSet** שנקרא changes. השיטה **GetErrors** של האובייקט **DataTable** מחזירה מערך שמורכב מכל השורות בטבלה שמכילות שגיאה אחת או יותר. אם אין שגיאות כלל, השיטה **GetError** מחזירה מערך ריק.

4. החלף את ההערה // If no error then update the database, other wise tell the user עם בלוק הקוד הבא.

יש מספר דרכים לדווח על השגיאות למשתמש. אחת הדרכים השימושיות ביותר היא לאתר את כל השגיאות ולדווח עליהן בהודעה אחת, שעשויה גם להיות ארוכה למדי.

```
if (badRows.Length == 0)
{
   // Update the database
}
else
{
   // Find the errors and inform the user
}
```

5. החלף את ההערה Find the errors and inform the user// עם המשפטים הבאים:

```
string errorMsg = null;
foreach (DataRow row in badRows)
{
   foreach (DataColumn col in row.GetColumnsInError())
   {
      errorMsg += row.GetColumnError(col) + "\n";
   }
}
MessageBox.Show("Errors in data: " + errorMsg,
      "Please fix", MessageBoxButtons.OK,MessageBoxIcon.Error);
```

קוד זה מדפדף בכל אחת משורות המערך **badRows**. בכל שורה עלולות להיות שגיאה אחת או יותר, והשיטה **GetColumnsError** מחזירה אוסף המורכב מכל העמודות שיש בהן שגיאה אחת לפחות. השיטה **GetColumnsError** אוספת את הודעות השגיאה של כל אחת מהעמודות למחרוזת אחת. כל אחת מהודעות השגיאה מצורפת למחרוזת errorMsg. לאחר בדיקת כל השורות והעמודות השגויות, היישום מציג תיבת הודעה עבור כל השגיאות. המשתמש צריך להיעזר במידע זה כדי לתקן את העדכונים שביצע ולשלוח אותם שוב.

כללי אחדות ואובייקטים מסוג DataSet

דוגמה זו מפעילה אובייקט **DataSet** פשוט, ולכן ייתכן שהשיטה **GetError** ככל הנראה לא תחזיר שגיאות כלל. בעת יצירת המחלקה **northwindDataSet**, נכלל בה מידע מבסיס הנתונים על עמודות המפתח העיקריות, סוגי הנתונים בכל עמודה, **כללי אחדות** (integrity rules) ועוד. לרוב, המשתמש מעדכן בשורות על ידי שימוש בפקד DataGridView שיש לו מספר מנגנוני בדיקת תקינות משלו, כפי שכבר ראינו בפרק זה. עם זאת, צריך לבדוק את תקינות הנתונים באובייקט **DataSet**, כי ייתכן שהפקד **DataGridView** אינו מצליח לתפוס את כל השגיאות.

לדוגמה נניח שהאובייקט **DataSet** מורכב ממספר טבלאות שהיחס ביניהן הוא של foreign key/primary key, ואנו לא רוצים לכפות על המשתמש להזין את הנתונים בסדר מסוים. במצבים כאלה צריך להשתמש בשיטה GetErrors. השיטה פועלת על בסיס ההנחה שהמשתמש סיים להזין את כל הנתונים, ולכן היא יכולה לערוך בדיקות מורכבות ולהצליב נתונים של שתי DataTables או יותר.

בסופו של דבר, לא ניתן לחזות מתי המשתמש יקליד משהו בלתי שגרתי במיוחד, ולכן רצוי תמיד לכתוב קוד שמגן בפני שגיאות כלליות. צריך ללכוד שגיאות בכל שלב שבו הן עשויות להופיע.

עדכון בסיס הנתונים

1. לאחר שווידאת את תקינות הנתונים, תוכל לשלוח אותם אל בסיס הנתונים. מצא את ההערה Update the database // אשר בשיטה **saveButton_Click** והחלף אותה במשפטים הבאים:

```
int numRows = productsTableAdapter.Update(changes);
MessageBox.Show("Updated " + numRows + " rows", "Success");
northwindDataSet.AcceptChanges();
```

קוד זה שולח את העדכונים באמצעות השיטה **Update** של האובייקט **productsTableAdapter**. לאחר עדכון השינויים, המשתמש מקבל הודעה על מספר השורות שעודכנו, והשיטה **AcceptChanges** מסמנת את השינויים כקבועים באובייקט DataSet. שים לב שהקוד נמצא בתוך בלוק **try/catch** כדי שיהיה ניתן לטפל בשגיאות.

חשוב

אם משתמש אחר שינה כבר את אחת השורות, או יותר, שבכוונתך לעדכן, השיטה **Update** תזהה את המצב המיוחד ותזרוק חריג. עליך להחליט כיצד אתה רוצה לטפל בחריג זה ביישום. לדוגמה, תוכל לאפשר למשתמש לעדכן את בסיס הנתונים בכל מקרה, למרות הסתירה בשינויים, או שתוכל לא לעדכן את הנתונים הסותרים ולרענן את האובייקט **DataSet** על ידי הנתונים החדשים שבבסיס הנתונים.

2. בנה והפעל את התוכנית. בטופס Products Maintenance, שנה את הערכים שבעמודות ProductName ו-SupplierID שבשתי השורות הראשונות, ולחץ Save.

השינויים שביצעת מעודכנים בבסיס הנתונים והיישום מודיע לך על עדכון שתי שורות.

3. סגור את הטופס והפעל שוב את היישום.

שם מוצר חדש וספק חדש מופיעים בשתי השורות הראשונות. השינויים אכן נשמרו בבסיס הנתונים.

4. סגור את הטופס וחזור לסביבת הפיתוח Visual Studio 2005.

☯ **אם ברצונך להמשיך לפרק הבא**

השאר את Visual Studio 2005 פעילה ועבור לפרק 25.

☯ **אם ברצונך לכבות כעת את Visual Studio 2005**

פתח את תפריט Menu ולחץ Exit. אם מופיעה תיבת דו-שיח Save, לחץ Yes.

פרק 24 – טבלה מסכמת

המשימה	צריך
להגדיר אובייקט **DataSet** חדש.	בתפריט Project בחר Add New Item ואחר כך בחר בתבנית DataSet. (יש להוסיף את הסיומת "xsd" לשם הקובץ).
להוסיף טבלאות נתונים (**DataTable**) ומתאמים (**TableAdapter**) לאובייקט **DataSet**.	פתח את החלון DataSet Designer. היעזר בערכת הכלים כדי להוסיף **TableAdapter** לאובייקט **DataSet**. היעזר באשף TableAdapter Configuration כדי להגדיר את השיטות של טבלאות הנתונים והמתאמים.
להשתמש בכריכה פשוטה (binding simple) כדי לכרוך מאפיין של פקד למקור מידע בסביבת העיצוב.	הרחב את המאפיין **DataBindings** של הפקד. בחר במאפיין שאתה רוצה לכרוך ובחר במקור המידע שאתה רוצה להשתמש בו.
להשתמש בכריכה פשוטה כדי לכרוך מאפיין של פקד למקור מידע בזמן ריצה.	השתמש בשיטה Add של המאפיין DataBindings. ציין את המאפיין שאתה רוצה לכרוך ואת מקור המידע. לדוגמה, כדי לכרוך את המאפיין Text של פקד הלחצן 1nottub למאפיין Text של הפקד textBox1, עליך להשתמש בקוד הבא: `myLabel.DataBindings.Add("Text", textBox1, "Text");`
להשתמש בכריכה מורכבת (complex binding) כדי לכרוך פקדים מסוג **ListBox**, **ComboBox** או **CheckedListBox** אל רשימת ערכים במקור המידע.	קבע את המאפיין **DataSource** של הפקד. ציין את הנתונים שברצונך להציג באמצעות המאפיין **DisplayMember**, ואחר כך ציין את הנתון שישמש כערכו של הפקד באמצעות המאפיין **ValueMember**.
להשתמש בכריכה מורכבת כדי לכרוך פקד **DataGridView** לרשימת ערכים במקור המידע.	קבע את המאפיין **DataSource** של הפקד. תוכל גם לציין את הנתונים שיוצגו וייערכו על ידי המשתמש, אולם אינך חייב לעשות זאת.
להגביל את הקלט המתקבל מהמשתמש באמצעות הפקד **DataGridView**.	לחץ לחיצה ימנית על הפקד **DataGridView** בחלון תצוגת העיצוב ומתפריט הקיצור בחר Edit Columns. היעזר בתיבת הדו-שיח Edit Columns כדי לקבוע את המאפיינים **ColumnType** ו-**ReadOnly** של הפקד. אם במאפיין **Column Type** תבחר **DataGridViewComboBox**, עליך לקבוע את המאפיינים **DataSource**, **DisplayMember** ו-**ValueMember** כפי שהיית עושה בכריכה מורכבת רגילה.

המשימה	צריך
לבדוק תקינות קלט המתקבל מהמשתמש דרך פקד **DataGridView**.	השתמש באירוע **CellValidating** כדי לבצע בדיקת תקינות כאשר המשתמש משנה את אחד התאים ומנסה לעבור לתא הבא. השתמש באירוע **CellEndEdit** כדי למחוק את הודעות השגיאה לאחר שבדיקת התקינות של הנתון החדש עברה בהצלחה. השתמש באירוע **DataError** כדי לטפל בשגיאות שחמקו מהאירועים האחרים.
לבדוק תקלות עקב שינויים שביצע המשתמש באובייקט **DataSet**.	קרא לשיטה **GetErrors** עבור כל טבלת נתונים שבאובייקט **DataSet**. דפדף בכל העמודות המכילות שגיאה, בכל אחת מהשורות, באמצעות השיטה **GetColumnsError**. בכל עמודה כזו קרא לשיטה **GetColumnError** כדי להציג את פרטי השגיאה שבעמודה.
לעדכן את מאגר הנתונים על פי המידע שבאובייקט **DataSet**.	הפעל את השיטה **Update** של האובייקט **TableAdapter** והעבר לה את האובייקט **DataSet** בתור פרמטר. אם העדכון עובר בהצלחה, הפעל את השיטה **AcceptChanges** על האובייקט **DataSet**; אם העדכון נכשל, קרא לשיטה **RejectChanges**.

חלק ו'

בניית יישומי Web

בחלק זה:

פרק 25

מבוא ל–ASP.NET

כאשר תסיים פרק זה, תוכל:

- ליצור דפים פשוטים על פי המודל של ASP.NET של Microsoft.
- לבנות יישומים שהגישה אליהם מתאפשרת באמצעות **דפדפן** אינטרנט (Web Browser).
- להשתמש באופן יעיל בפקדי שרת של ASP.NET.
- לשלוט בעיצוב דפי האתר על ידי שימוש **בערכות נושאים** (Themes).

בחלקים הקודמים של ספר זה למדת לבנות יישומי #C הפועלים בסביבת שולחן העבודה של Microsoft Windows. יישומים אלה מספקים בדרך כלל למשתמש גישה לבסיסי נתונים באמצעות ADO.NET. בחלק האחרון של הספר נעסוק בעולם יישומי הרשת. אלה הם יישומים שניתן לגשת אליהם דרך האינטרנט, כאשר הממשק אינו תצורת שולחן העבודה הרגילה, אלא דפדפן האינטרנט, ובקצרה - **דפדפן**.

בשלושת הפרקים הראשונים של חלק זה תבחן את המחלקות הקיימות ב- NET Framework שבאמצעותן ניתן לבנות יישומי רשת. תלמד על המבנה של ASP.NET וגם על **טפסי רשת** (Web forms) ו**פקדי שרת** (Server control). תראה שהמבנה של יישומים המופעלים ברשת שונה מהמבנה של יישומים הפועלים בסביבת שולחן העבודה, ותלמד לבנות אתרי אינטרנט יעילים ופשוטים לאחזקה.

בפרק האחרון שבחלק זה תלמד על **שירותי רשת** (Web services), אשר מאפשרים לבנות **יישומים מבוזרים** (distributed applications) המורכבים מרכיבים ושירותים המפוזרים ברחבי האינטרנט (או אינטראנט). תלמד ליצור שירותי רשת ותבין כיצד שירותי רשת בנויים על בסיס SOAP (Simple Object Access Protocol). תלמד גם מספר טכניקות שתוכל ליישם כדי לחבר **יישומי לקוח** (client application) אל שירות רשת רצוי.

האינטרנט כתשתית

האינטרנט היא למעשה רשת גדולה מאוד, חובקת עולם, וכתוצאה מכך - המידע והנתונים שאתה ניגש אליהם באמצעותה, עשויים להיות מרוחקים מאוד מתחנת העבודה שלך. עובדה זו צריכה להשפיע על אופן העיצוב של היישומים. לדוגמה, ייתכן שתוכל לנעול את הנתונים שאתה שולף מבסיס הנתונים כאשר אתה עוסק ביישום קטן המיועד לשימוש בסביבת שולחן העבודה, אולם הדבר אינו אפשרי ביישומים שהגישה אליהם מתבצעת דרך האינטרנט לאין-ספור אנשים. ניצול המשאבים הוא גורם חשוב הרבה יותר וגם קריטי באינטרנט, לעומת יישומים מקומיים.

רוחב-פס (bandwidth) של הרשת הוא משאב נדיר שלעצמו, ויש להשתמש בו בתבונה ובחסכנות. הביצועים של הרשתות המקומיות מושפעים בדרך כלל משעת השימוש במשך היום (לדוגמה, עומס חריג ביום חמישי אחר הצהריים כאשר כולם מנסים לסיים את העבודה לפני סוף השבוע), מסוג היישומים ומכלול של גורמים נוספים. אם ביצועי הרשתות המקומיות מושפעים בקלות שכזו, קל וחומר שרשת האינטרנט תהיה אפילו יותר בלתי צפויה. בעת שימוש באינטרנט אתה תלוי במספר שרתים אשר מנתבים את בקשותיך מהדפדפן אל האתר המבוקש, והמענה מנותב גם הוא באותו נתיב, או בנתיב אחר, בכיוון ההפוך. הפרוטוקולים ומנגנוני העברת הנתונים שעל בסיסם בנויה רשת האינטרנט, משקפים את העובדה שרשתות עשויות להיות בלתי אמינות (ולרוב הן באמת כך), וגם בלתי צפויות. בכל יישום הפועל באמצעות שרת, עשויים להשתמש מספר רב של משתמשים אשר מפעילים מספר רב של דפדפנים הפועלים על מספר רב של מערכות הפעלה, והדבר יכול להיות מקור לחוסר יציבות.

בקשות ותגובות של שרתי רשת

משתמש שניגש אל יישום דרך האינטרנט באמצעות דפדפן, משתמש ב-HTTP (HyperText Transfer Protocol) כדי ליצור קשר עם היישום. היישומים נמצאים בדרך כלל על סוג כלשהו של שרת רשת אשר קורא בקשות HTTP ומחליט באיזה יישום להשתמש כדי לענות לבקשה. בהקשר הזה, למונח **יישום** (application) אין הגדרה חד-משמעית - השרת עשוי להפעיל תוכנית אשר תבצע פעולה כלשהי, או שהוא עשוי גם לעבד את הבקשה בעצמו על פי היגיון פנימי משלו, או להפעיל אמצעים אחרים. בכל אופן, הבקשה מעובדת ושרת הרשת שולח מענה ללקוח על ידי שימוש בפרוטוקול HTTP. תוכן התשובה של HTTP תהיה לרוב דף HTML (HyperText Markup Language); רוב הדפדפנים מבינים את שפת HTML ויודעים להציג דפים בשפה זו.

הערה

יישומים המשמשים להפעלת יישומים אחרים דרך האינטרנט נקראים לעיתים קרובות **לקוח** (client) או **יישומי לקוח**. היישומים שהם מפעילים, נקראים לעיתים קרובות **שרתים** (servers) או **יישומי שרת**.

מצב הניהול

HTTP הוא פרוטוקול **ללא-חיבור** (connectionless). כלומר, כל בקשה (או מענה) הוא צרור נתונים העומד בפני עצמו. תקשורת ממוצעת בין לקוח ליישום הפועל בשרת רשת, כוללת מספר בקשות. לדוגמה, המשתמש יכול להציג דף אינטרנט, להזין נתונים, ללחוץ על מספר לחצנים. התצוגה תשתנה כפי הצורך כדי לאפשר למשתמש להקליד נתונים נוספים, וחוזר חלילה. כל בקשה שנשלחת על ידי המשתמש לשרת מופרדת מהבקשות האחרות שנשלחו על ידי המשתמש, או על ידי משתמשים אחרים המשתמשים באותו שרת (או מפעילים את אותו יישום) באותו זמן. הבעיה היא, שלעיתים קרובות בקשה של לקוח חייבת להופיע **בהקשר** (context) מסוים או **במצב** (state) מסוים.

ניקח לדוגמה את התרחיש השכיח הבא. יישום רשת מאפשר למשתמש לעיין במוצרים שונים המוצעים למכירה. אם המשתמש רוצה לקנות פריטים אחדים, עליו להניח אותם בעגלת קניות וירטואלית. אחד האלמנטים השימושיים ביישום זה היא היכולת להציג את התוכן הנוכחי של העגלה.

היכן כדאי לאחסן את תוכן עגלת הקניות (המצב של הלקוח)? אם מידע זה יאוחסן בשרת, אז השרת יצטרך לבדוק את כל בקשות HTTP ולשייך כל אחת מהן לתוכן העגלה הרלוונטית. הדבר אפשרי, אולם משמעות הדבר היא עיבוד נתונים נוסף שמטרתו התאמה בין הבקשות השונות למצבי הלקוח השונים. פעולה זו מחייבת שימוש בבסיס נתונים לשימור נתוני המצב בין בקשה לבקשה. הבעיה בטכניקה הזו היא שלשרת אין אחריות, כי לאחר שמירת נתוני המצב, המשתמש עשוי לשלוח בקשה נוספת לשימוש בנתונים או למחיקתם. אם שרת הרשת ישמור את כל נתוני המצב של כל הלקוחות המשתמשים בו, ייתכן שיצטרך לתפעל בסיס נתונים גדול מאוד!

החלופה היא לאחסן את נתוני המצב במחשב של המשתמש. **הפרוטוקול Cookies** הומצא כדי לאפשר לשרתי רשת לאחסן נתונים בתור cookies (עוגיות) במחשב של המשתמש. החסרונות של גישה זו הם שהיישום צריך לשלוח את הנתונים שבקבצים הללו דרך הרשת כחלק מכל בקשת HTTP לשימושו של השרת. בנוסף, היישום צריך לוודא שהקבצים אינם חורגים מגודל מסוים. כנראה שהחיסרון העיקרי של הפרוטוקול cookies הוא בכך שהמשתמש יכול לבטל את השימוש בפרוטוקול, ולמנוע מהשרת לאחסן קבצי cookie במחשב שלו, וכך להכשיל את ניסיון השמירה של נתוני המצב.

השירותים של ASP.NET

על פי מה שלמדנו עד כה, התשתית לבנייה ולהפעלה של יישומי רשת חייבת להתייחס למספר סוגיות, משמע, עליה לעמוד בדרישות הבאות:

- תמיכה בפרוטוקול HTTP סטנדרטי.
- ניהול מצב הלקוח באופן יעיל.
- מתן כלים המאפשרים פיתוח יישומי רשת בקלות יחסית.
- יצירת יישומים שהגישה אליהם מתאפשרת באמצעות כל דפדפן רשת התומך ב-HTML.
- היישומים צריכים להגיב לבקשות הלקוח.

מיקרוסופט פיתחה את המודל Active Server Pages או ASP, כתגובה לחלק גדול מהסוגיות שעסקנו בהן. מודל ASP איפשר למפתחי היישומים לשלב את הקוד של היישום בדפי HTML. שרת רשת כגון IIS (Internet Information Services) יכול להפעיל את הקוד של היישום ולהשתמש בו כדי לחולל תגובה בפורמט של HTML. במודל זה היו מספר בעיות: לביצוע פעולות פשוטות, כגון הצגת דף נתונים מבסיס נתונים, היה צורך בכתיבת כמות גדולה של קוד; שילוב של קוד רגיל וקוד HTML הקשה על ההבנה והאחזקה של היישום; לעיתים קרובות הביצועים לא היו במיטבם, מכיוון שהמודל ASP צריך היה לתרגם את הקוד של בקשת HTML בכל פעם שהבקשה נשלחה, גם כאשר הופיע קוד זהה בכל פעם.

כאשר פורסמה טכנולוגיית .NET., מיקרוסופט עדכנה את ASP ויצרה את ASP.NET. אלה הם הרכיבים העיקריים של ASP.NET:

- **מודל תכונות לוגי** הכולל טפסי Web עבור הלוגיקה שברקע של התצוגה, וקבצי קוד עבור הלוגיקה העסקית. ניתן לכתוב קוד בכל השפות הנתמכות על ידי .NET., וביניהן #C. טפסי Web של ASP.NET עוברים הידור ומוטמנים בשרת הרשת כדי לשפר את הביצועים.
- **פקדי שרת** אשר תומכים באירועים (events) מצד השרת, אך מעוצבים בפורמט HTML כדי שיוכלו לפעול בצורה חלקה בדפדפן הרשת (תומך HTML). מיקרוסופט שיפרה גם מספר רב של פקדי HTML קיימים כדי שתוכל לשלוט בהם דרך הקוד.
- **פקדי נתונים** המשמשים להצגה, עריכה ואחזקה של נתונים בבסיס נתונים.
- **אפשרויות לשמירת מצב הלקוח במטמון** שבמחשב שלו באמצעות cookies, בשירות מיוחד (State server של ASP.NET) שנמצא בשרת הרשת, או בבסיס הנתונים SQL Server של מיקרוסופט. ניתן להשתמש בקוד כדי לתכנת בקלות יחסית את זיכרון **המטמון** (cache).

במהדורה האחרונה של Net Framework, הוסיפה Microsoft שיפורים למודל ASP.NET. השיפורים נועדו לשפר את עיבוד הנתונים המתקבלים דרך הרשת ובאחזקה של אתרים. מיקרוסופט הוסיפה את האלמנטים הבאים:

- **הכלים Master Pages**, **Themes** ו- **Web Parts** אשר משפרים את יכולות עיצוב הדפים ועריכת הדפים. Master Pages (**דפי תבנית**) משמשים לקביעת עיצוב משותף לכל הדפים ביישום מסוים. Themes (**ערכות נושא**) משמשים להקניית מראה ותחושה אחידה בכל דפי האתר, ובמידת הצורך הם גם מאפשרים לוודא שכל הפקדים יהיו זהים זה לזה. Web Parts (**רכיבי אינטרנט**) מאפשרים ליצור דפי Web מודולריים המשתנים על פי דרישות המשתמש. בספר זה לא נעסוק בכלים Master Pages ו- Web Parts.
- **פקדי מקור נתונים** (data source control), חדשים המשמשים לכריכת נתונים לדפי Web. הפקדים החדשים מאפשרים לבנות יישומים שיכולים להציג ולערוך נתונים בקלות ובמהירות. פקדי מקור נתונים מתאימים למגוון רחב של מקורות מידע, וביניהם SQL Server, Microsoft Access, קבצי XML, שירותי Web ואובייקטים עסקיים שיכולים להחזיר נתונים. השימוש בפקדי מקור נתונים היא דרך טובה לפעול על נתונים, מבלי להיות תלויים במקור שלהם. בפרק 27 תשתמש בפקדי מקור נתונים.
- **פקדים חדשים ומשופרים**. מיקרוסופט יצרה את הפקדים GridView, DetailsView ו-FormView להצגה ועריכה של נתונים. באפשרותך להשתמש בפקד TreeView להצגת נתונים המסודרים בסדר היראר כי, ובאמצעות הפקדים SiteMapPath ו-Menu תוכל לסייע למשתמש לנווט ביישומי הרשת שתבנה. בפקד GridView תשתמש בפרק 27.

- **מנגנוני אבטחה משופרים** כוללים תמיכה מובנית באימות ובאישור משתמשים. באפשרותך להתיר למשתמשים גישה ליישומי הרשת שלך, לבצע בדיקות הרשאה למשתמשים כאשר הם מנסים להירשם (log in), ולחקור את נתוני המשתמש, כדי לדעת מי מנסה לגשת לאתר. תוכל להשתמש בפקד Login כדי לבקש מהמשתמש אמצעי זיהוי ולאשר אותו, או להשתמש בפקד Password Recovery כדי לעזור למשתמשים להיזכר או לאתחל את הסיסמה שלהם. את השימוש בפקדי האבטחה תתרגל בפרק 27.
- **בנייה משופרת וניהול של אתרים** על ידי שימוש בכלי ASP.NET Web Site Administration Tool. כלי זה כולל אשפים המשמשים לקביעת התצורה ואבטחתם של יישומי Web של ASP.NET. בכלי ASP.NET Web Site Administration Tool תשתמש בפרק 27.

בהמשך פרק זה תלמד עוד על מבנה היישומים של ASP.NET.

יצירת יישומי רשת בעזרת ASP.NET

יישום רשת המשתמש במודל של ASP.NET מורכב בדרך-כלל מדף אחד או יותר של ASP.NET (Web forms), קבצי קוד וקבצי תצורה.

טופס Web מאוחסן בקובץ aspx, אשר מהווה בעיקרון קובץ HTML רגיל שנוספו לו מספר תגיות ייחודיות של Microsoft.NET. קובץ aspx מגדיר את העיצוב והמראה של דף, ויש לו לעיתים קרובות גם קובץ נלווה המכיל קוד, שמגדיר את הלוגיקה של היישום עבור הרכיבים שבקובץ aspx, כגון מטפלים באירועים (event handler) ושיטות עזר. **תגית** (**tag**), או הנחיה, בראש כל קובץ aspx מציינת את השם והמיקום של קובץ הקוד הרלוונטי. המודל של ASP.NET תומך גם באירועים ברמת היישום, אשר מוגדרים בקבצי Global.aspx.

כל יישום רשת יכול לכלול גם קובץ תצורה בשם Web.config. קובץ זה, בפורמט XML, מכיל מידע על אבטחה, ניהול מטמון, הידור הדפים וכו'.

בניית יישום ASP.NET

בתרגיל הבא עליך לבנות יישום ASP.NET פשוט שמפעיל פקדי Server כדי לאסוף מהמשתמש את נתוני העובדים המועסקים בחברת תוכנה כלשהי. בתרגיל זה תלמד כיצד בנוי יישום רשת ממוצע.

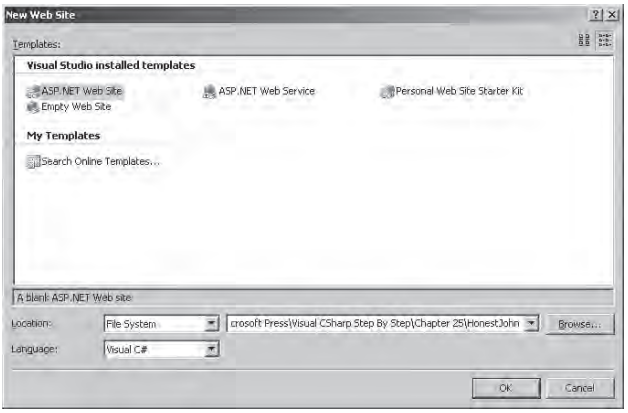
הערה

אינך חייב להפעיל במחשב שלך את שרת האינטרנט (IIS) כדי לפתח יישומי רשת, כי סביבת הפיתוח Visual Studio 2005 כוללת עבורך שרת פיתוח (Development Web server) מיוחד. כאשר אתה בונה ומפעיל יישום רשת, ברירת המחדל של סביבת הפיתוח היא להפעיל את היישום באמצעות השרת המובנה שלו.

בניית יישום רשת

1. הפעל את Microsoft Visual Studio 2005 אם אינה פועלת.
2. בתפריט File, New בחר Web Site.

   על המסך תופיע תיבת הדו-שיח New Web Site.
3. לחץ על התבנית ASP.NET Web Site. ברשימה הנפתחת Location בחר File System, והקלד **C:\Documents and Settings**\YourName**My Documents\Microsoft Press\Visual CSharp Step by Step\Chapter 25\HonestJohn**. את **YourName** החלף בשם המשתמש שלך ב-Windows. בשדה Language שיוצג בחר Visual C# ולחץ OK.



הערה

ברשימה Location עליך לבחור File System (מערכת קבצים) כדי להשתמש בשרת Development Web server של סביבת הפיתוח 2005. כדי להשתמש ב-IIS עליך לשנות את Location ל-HTTP ולציין את הכתובת (URL) של האתר שברצונך להקים, ולא את שם הקובץ.

כעת ייווצר יישום המורכב מ**תיקיית רשת** (Web Folder) בשם App_Data, וטופס Web בשם Default.aspx. הקוד HTML עבור דף ברירת מחדל זה יופיע בחלון **תצוגת המקור** (Source View).

4. ב- Solution Explorer בחר בקובץ Default.aspx. בחלון Properties, שנה את המאפיין File Name של הקובץ Default.aspx ל-EmployeeForm.aspx.
5. לחץ על הלחצן Design אשר בתחתית הטופס משמאל, כדי לעבור ל**תצוגת העיצוב** (Design View) של הטופס. נכון לעכשיו, חלון תצוגת העיצוב אינו מציג דבר.

   בחלון תצוגת העיצוב ניתן לגרור פקדים מערכת הכלים אל טופס Web, ולאחר מכן סביבת הפיתוח תיצור עבורך באופן אוטומטי את קוד HTML של הפקדים. קוד HTML זה מופיע בחלון תצוגת המקור כאשר אתה משתמש בו כדי לצפות בטופס. ניתן לערוך את קוד HTML גם באופן ידני.

בתרגיל הבא עליך להגדיר סגנון עבור הטופס ולהוסיף לו פקדים כדי שיהיה שימושי. שימוש ב-style מאפשר לוודא שכל הפקדים בטופס יהיו בעלי מראה ותחושה משותפים (מבחינת הצבע והגופן, למשל), וגם מאפשר לשנות מאפיינים אחרים, כגון תמונת הרקע של הטופס.

הערה

מיקרוסופט שחררה תוסף (add in) בשם Web Application Project עבור סביבת הפיתוח, שמטרתו לספק מודל הדומה לגרסה הקודמת לבניית יישומי אינטרנט (Visual Studio 2003) ואשר מקלה על מעבר של מערכות Web מהגרסה הקודמת לגרסת Visual Studio 2005. בנוסף, מיקרוסופט הכניסה שיפורים שונים בתגובה למשוב מהלקוחות (למידע נוסף, עיין במאמר Introduction to Web Application Projects בתיעוד של MSDN).

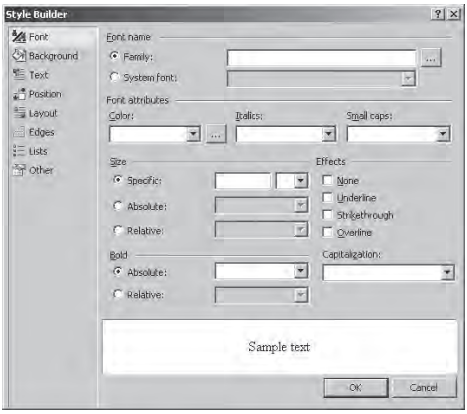
עיצוב טופס Web

1. לחץ על הטופס בחלון תצוגת העיצוב. בחלון Properties, שנה את המאפיין Title של האובייקט DOCUMENT ל- **Employee Information**.

   הערך שתציב במאפיין Title יופיע בכותרת של דפדפן הרשת כאשר תשתמש בו להפעלת יישום הרשת.

2. בחר במאפיין Style ולחץ על לחצן ההמשך (...).

   כתוצאה, תיפתח תיבת הדו-שיח Style Builder. תיבת דו-שיח זו מאפשרת לעצב את הסגנון (**style**) של הטופס (הסגנון קובע את הצבע, הגופן, העיצוב ומאפייני ברירת מחדל אחרים של הטופס ופקדיו השונים).



3. באזור Font Name שבטופס, ודא שתיבת הסימון Family מסומנת, ולאחר מכן לחץ על לחצן ההמשך שמימינה. בתיבת הדו-שיח Font Picker שתיפתח, בחר בגופן Ariel ברשימת הגופנים המותקנים Installed Fonts, ולחץ על הלחצן << כדי להוסיף אותו לרשימה של הגופנים שנבחרו, Selected Fonts. לחץ OK כדי לחזור לתיבת הדו-שיח Style Builder.

4. ברשימה הנפתחת Color בחר Blue.

5. בחלונית השמאלית של תיבת הדו-שיח לחץ Background. בדף Background שיופיע סמן את תיבת הסימון Transparent (שקיפות).

6. השתמש ב- Windows Explorer כדי להעתיק את הקובץ \My Documents Microsoft Press\Visual CSharp Step By Step\Chapter 25\computer.bmp אל התיקייה \Microsoft Press\Visual CSharp Step By Step\Chapter 25\HonestJohn.

7. חזור לתיבת הדו-שיח Style Builder שבסביבת הפיתוח Visual Studio 2005. בתיבת הטקסט Image הקלד **computer.bmp** ולחץ OK.

8. פתח את ערכת הכלים והרחב (במידת הצורך) את הקטגוריה Standard.

ערכת הכלים מכילה פקדים שניתן לגרור אל טפסי ASP.NET. בחלק גדול מהמקרים, פקדים אלה דומים לפקדים שהשתמשת בהם לבניית טפסי Windows. ההבדל הוא בכך שפקדים אלה עוצבו במיוחד לשימוש בסביבת HTML, והמימוש שלהם הוא באמצעות HTML בסביבת ההרצה.

9. גרור מערכת הכלים ארבעה פקדי Label ושלושה פקדי TextBox אל טופס ה-Web. שים לב שהגופן והצבע של הפקדים יהיה תואם לסגנון שהגדרת באמצעות תיבת הדו-שיח Style Builder.

הערה

הפקדים מוצבים באופן אוטומטי משמאל לימין בחלון תצוגת העיצוב. אל תדאג בקשר למיקום הפקדים, כי ניתן להציב אותם בכל מקום לאחר קביעת מאפייניהם.

הערה

מלבד השימוש בפקד **Label**, ניתן להוסיף כיתוב בטופס Web. אולם קשה לקבוע את התצורה ולשנות את המאפיינים של טקסט כזה וקשה גם להחיל עליו ערכות (Themes). כאשר עליך לבנות אתר שתומך גם בשפות אחרות (כגון צרפתית, אנגלית או עברית), עדיף להשתמש בפקדי **Label**, מכיוון שכך יהיה קל יותר להתאים לשפה (localize) את הכיתוב המוצג באמצעות קבצי Resource. למידע נוסף בנושא זה קרא את "Resources in Applications" שב- Microsoft Visual Studio 2005 Documentation.

10. בחלון Properties שנה את מאפייני הפקדים שהוספת על פי הערכים שבטבלה הבאה:

Control	Property	Value
Label1	Font Bold (expand the Font property)	**True**
	Font Name	**Arial Black**
	Font Size	**X-Large**
	Text	**Honest John Software Developers**
	Height	**36px**
	Width	**630px**
Label2	Text	**First Name**
Label3	Text	**Last Name**

Control	Property	Value
Label4	Text	**Employee Id**
TextBox1	Height	**24px**
	Width	**230px**
	(ID)	**firstName**
TextBox2	Height	**24px**
	Width	**230px**
	(ID)	**lastName**
TextBox3	Height	**24px**
	Width	**100px**
	(ID)	**employeeID**

11. בחלון תצוגת העיצוב סמן את כל ארבע התוויות ואת שלוש תיבות הטקסט (לחץ על **Label1**, ולאחר מכן לחץ על כל אחד מהפקדים האחרים בעודך מחזיק את מקש **Shift** לחוץ).

12. בתפריט Layout, Position בחר Absolute.

כעת תוכל לגרור את הפקדים למקום אבסולוטי בטופס במקום שסביבת הפיתוח תציב אותם עבורך באופן אוטומטי.

13. מקם את התוויות ואת תיבות הטקסט על פי המסך הבא:

טיפ

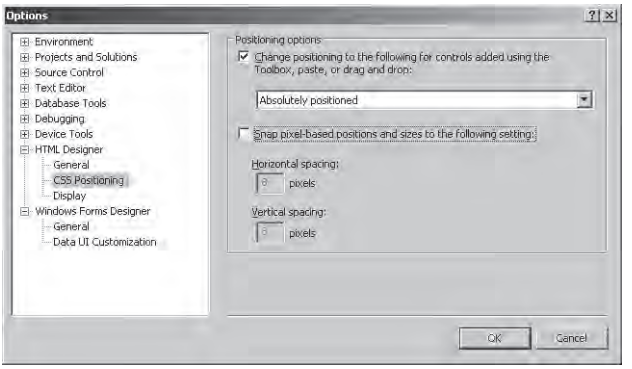
ניתן ליישר את הפקדים ואת המרווחים ביניהם על ידי הפקודות שבתפריט Format. כדי ליישר קבוצת פקדים, עליך לסמן את כולם, לפתוח את תפריט Format, ללחוץ Align ולבחור בסוג היישור הרצוי (Lefts, Centers, Rights, Tops, Middles או Bottoms). תוכל גם ליצור מרווח אחיד בין הפקדים על ידי סימונם והפעלת הפקודה Horizontal Spacing או Vertical Spacing מתפריט Format.

14. לחץ על לחצן Source בתחתית הטופס, להצגת קוד HTML של הטופס בחלון תצוגת המקור. קוד HTML צריך להיראות דומה לקוד הבא (מיקום הפקדים עשוי להיות שונה במקצת בטופס שלך):

```
<%@ Page Language="C#" AutoEventWireup="true"
CodeFile="EmployeeForm.aspx.cs "Inherits="_Default" %>

<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.1//EN" "http://www.
w3.org/TR/xhtml11/DTD/xhtml11.dtd">

<html xmlns="http://www.w3.org/1999/xhtml" >
<head runat="server">
   <title>Employee Information</title>
</head>
<body style="background-image: url(computer.bmp); color: blue;
font-family: Arial;
background-color: transparent">
   <form id="form1" runat="server">
   <div>
      <asp:Label ID="Label1" runat="server" Text="Honest John
Software Developers"
style="z-index: 100; left: 96px; position: absolute; top: 24px"
Font-Bold="True" Font-Names="Arial Black" Font-Size="X-Large"
Height="36px" Width="630px"></asp:Label>
      <asp:Label ID="Label2" runat="server" Text="First Name"
style="z-index: 101;left: 62px; position: absolute; top:
104px"></asp:Label>
      <asp:Label ID="Label3" runat="server" Text="Last Name"
style="z-index: 102; left:414px; position: absolute; top:
104px"></asp:Label>
      <asp:Label ID="Label4" runat="server" Text="Employee
Id" style="z-index:103; left: 62px; position: absolute; top:
167px"></asp:Label>  
      <asp:TextBox ID="firstName" runat="server" style="z-index:
104; left: 156px;position: absolute; top: 101px" Height="24px"
Width="230px"></asp:TextBox>
      <asp:TextBox ID="lastName" runat="server" style="z-index:
107; left: 507px;position: absolute; top: 101px" Height="24px"
Width="230px"></asp:TextBox>
      <asp:TextBox ID="employeeID" runat="server" style="z-
index: 106; left: 160px;position: absolute; top: 161px"
Height="24px" Width="100px"></asp:TextBox>
   </div>
   </form>
</body>
</html>
```

15. חזור לחלון תצוגת העיצוב.

16. בתפריט Layout, Position בחר Auto-position Options. בתיבת הדו-שיח Options, הרחב את הרכיב HTML Designer שבפקד TreeView אם לא עשית זאת עדיין, ולחץ CSS Positioning. בחלונית שבצד הימני של תיבת הדו-שיח, סמן את תיבת הסימון "Change positioning to the following for controls added by using the Toolbox, paste, or drag and drop". בתפריט הנפתח בחר Absolutely positioned:



פעולה זו מאפשרת למקם את כל הפקדים שתוסיף לטופס בעתיד, בנקודה שבה תשחרר אותם על פני הטופס. אינך צריך לבחור עוד באפשרות Absolute (כלומר, מיקום קבוע) עבור כל אחד מהם.

17. הוסף פקד Label נוסף וארבעה פקדי RadioButton (לחצני אפשרויות) לטופס Web. קבע את מאפייני הפקדים החדשים על פי הנתונים המופיעים בטבלה הבאה:

Control	Property	Value
Label5	Text	**Position**
RadioButton1	Text	**Worker**
	TextAlign	**Left**
	GroupName	**positionGroup**
	Checked	**True**
	(ID)	**workerButton**
RadioButton2	Text	**Boss**
	TextAlign	**Left**
	GroupName	**positionGroup**
	Checked	**False**
	(ID)	**bossButton**
RadioButton3	Text	**Vice President**

Control	Property	Value
	TextAlign	**Left**
	GroupName	**positionGroup**
	Checked	**False**
	(ID)	**vpButton**
RadioButton4	Text	**President**
	TextAlign	**Left**
	GroupName	**positionGroup**
	Checked	**False**
	(ID)	**presidentButton**

המאפיין GroupName קובע כיצד לקבץ את לחצני האפשרויות. כל הלחצנים שערך המאפיין GroupName שלהם זהה, יהיו בקבוצה אחת. משמעות הדבר היא שרק אחד מהם יכול להיות מסומן בכל עת.

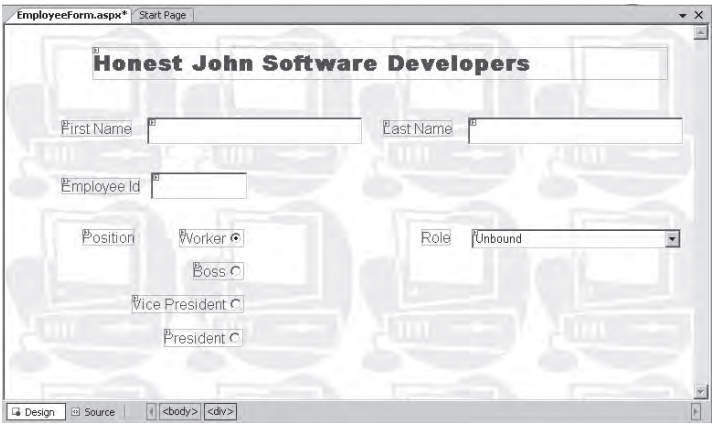
18. הצב את הפקדים החדשים על פי המסך שלהלן:

19. הוסף פקד Label ופקד DropDownList (רשימה נפתחת) לטופס Web. שנה את מאפייניהם על פי הנתונים שבטבלה הבאה:

Control	Property	Value
Label6	Text	Role
DropDownList1	Width	230px
	(ID)	positionRole

הרשימה הנפתחת **positionRole** מורכבת מהתפקידים השונים במפעל, והיא משתנה בהתאם למשרה שהעובד ממלא. תצטרך לכתוב קוד כדי למלא את הרשימה באופן דינמי.

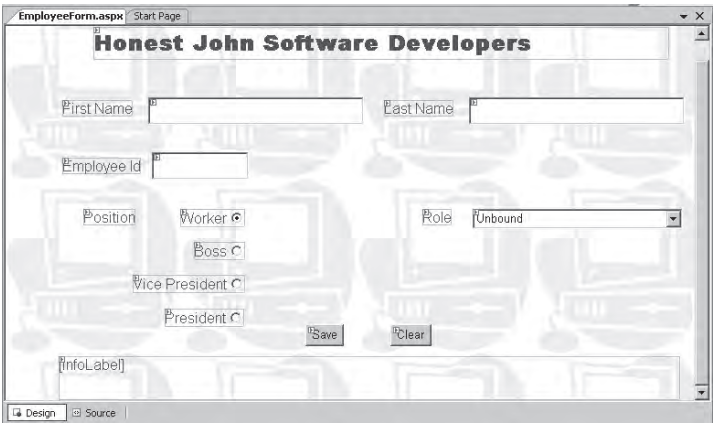
20. הצב את הפקדים החדשים על פי המסך הבא:



21. הוסף לטופס שני פקדי Buttons ופקד **Label**. שנה את מאפייניהם על פי הנתונים שבטבלה הבאה:

Control	Property	Value
Button1	Text	Save
	(ID)	saveButton
Button2	Text	Clear
	(ID)	clearButton
Label7	Text	leave blank
	Height	48px
	Width	680px
	(ID)	infoLabel

22. הצב את הפקדים החדשים על פי המסך הבא:



הלחצנים מציפים מטפלים באירועים (event handler), שתכתוב אותם באחד התרגילים שבהמשך. הלחצן Save אמור לאסוף את הנתונים שהמשתמש הזין, ולהציג אותם בפקד **InfoLabel** שבתחתית הטופס. הלחצן Clear אמור לאפס את תיבות הטקסט ואת הפקדים האחרים לערכי ברירת המחדל שלהם.

בדוק את הטופס Web

1. בתפריט Debug בחר Start Debugging.

Visual Studio 2005 תבנה את היישום, שרת Development של ASP.NET יופעל, והדפדפן Internet Explorer ייפתח ויציג את הטופס.

הערה

בפעם הראשונה שתפעיל יישום רשת באמצעות הפקודה Start Debugging תיבת הודעה תודיע שניפוי השגיאות אינו מופעל (debugging is not enabled). עליך לבחור "Run without debugging", או לבחור "Modify the Web.config file to enable debugging", אם ברצונך להפעיל את התוכנית במצב debug (ניפוי). להפעלת התוכנית במצב ניפוי קובעים נקודות שבירה וניתן לסקור את הקוד פקודה אחר פקודה, כפי שהוסבר בפרק 3. במצב ניפוי היישום פועל יותר לאט בגלל המטלות הנוספות. לכן, תצטרך לכבות את מצב הניפוי בעת העברת היישום לעבודת אמת באתר הרשת. כדי לעשות זאת, עליך להסיר מהקובץ Web.config את השורה הבאה: <compilation debug="true"/>

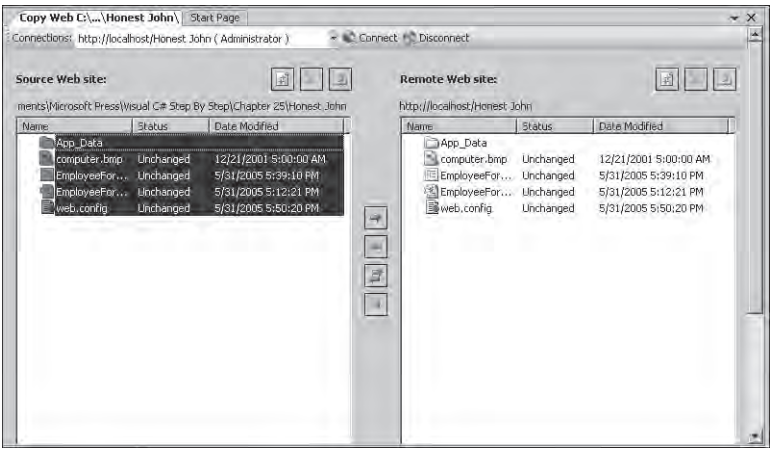
2. הזן נתונים של עובד פיקטיבי. נסה את לחצני האפשרויות כדי לוודא שלא ניתן לבחור ביותר מאחד מהם במקביל. לחץ על החץ של הרשימה הנפתחת Role; הרשימה תהיה ריקה. לחץ Save, Clear.

3. סגור את Internet Explorer וחזור לסביבת הפיתוח Visual Studio 2005.

התאמת האתר לשרת IIS

תפריט Website של Visual Studio 2005 מכיל את הפקודה Copy Web Site, המשמשת להעתקת אתרי אינטרנט ממקום אחד לאחר. ניתן להשתמש בפקודה זו כדי להתאים אתרים שנבנו ונבדקו על ידי השרת Development Server של ASP.NET לאתר הייצור של IIS (תחילה עליך ליצור אתר חדש או תיקייה וירטואלית ריקה באמצעות בקרת הניהול של Internet Information Services).

התחבר לתיקייה הווירטואלית באתר הייצור של IIS. כעת תוכל להעתיק קבצים בודדים באופן סלקטיבי אל אתר היצור וממנו, או לסנכרן את הקבצים בין מספר אתרים.



למידע נוסף בנושא ראה "Walkthrough: Copying a Web Site Using the Copy Web Site Tool" וגם "How to: Copy Web Site Files with the Copy Web Tool" שנמצאים ב- Visual Studio 2005 Documentation.

הכרת פקדי Server

קבוצת הפקדים שהוספת לטופס Web נקראת פקדי Server. פקדי שרת אלה דומים במקצת לפקדי HTML סטנדרטיים שניתן להוסיף לכל דפי האינטרנט, מלבד העובדה שהם יותר נוחים לתכנות. רוב פקדי Server משתמשים במטפלי אירוע, בשיטות ובמאפיינים שהקוד המופעל בשרת יכול לנצל ולשנות באופן דינמי בסביבת ההרצה. בתרגילים הבאים, תלמד כיצד לתכנת פקדי Server.

בחינת פקד Server

1. בסביבת הפיתוח Visual Studio 2005, הצג את הטופס EmployeeForm.aspx בחלון תצוגת המקור.

2. עיין בקוד HTML של הטופס. שים לב שהוא כולל את ההגדרות של כל אחד מהפקדים. אתר את הגדרת **התווית** הראשונה ובחן אותה לעומק (את הקוד שלהלן ערכנו באופן שיהיה יותר קריא, ולכן הוא אינו נראה בדיוק כפי שהוא מופיע בקובץ המקורי):

```
<asp:Label ID="Label1" runat="server"
   Text="Honest John Software Developers"
   style="z-index: 100; left: 96px;
   position: absolute; top: 24px"
   Font-Bold="True" Font-Names="Arial Black"
   Font-Size="X-Large" Height="36px" Width="630px">
</asp:Label>
```

עליך לשים לב לשתי נקודות חשובות. תחילה הבט בסוג הפקד: **asp:Label**. כל הפקדים של טפסי Web נמצאים במרחב השמות "asp", כי כך נקבע על ידי מיקרוסופט. אחר כך צריך לשים לב לתכונה (attribute) הזו: **"runat="server**. משמעותה שניתן לגשת לפקד על ידי תכנות של קוד שפועל בשרת. קוד זה יכול לחקור ולשנות את הערך של כל אחד ממאפייני הפקד (לדוגמה, לשנות את הכיתוב שלו).

הערה

כל הפקדים חייבים להסתיים בתגית <asp:**control**/>. את control צריך להחליף בסוג הפקד, כגון Label בדוגמה שלעיל.

3. חזור לחלון תצוגת העיצוב.

פקדי HTML

סביבת העבודה ASP.NET תומכת גם בפקדי HTML. כשתרחיב את הקטגוריה HTML שבערכת הכלים, תוצג רשימת פקדים שמיקרוסופט שילבה במודל המקורי של Active Server Pages. הם עדין מופיעים, כדי שתוכל להמיר דפי ASP לדפי ASP.NET בקלות יחסית. עם זאת, כאשר אתה מתחיל לבנות יישום חדש, הקפד להשתמש בפקדי Standard Web Forms בלבד.

גם פקדי HTML כוללים את התכונה **runat**, המאפשרת לבחור היכן להפעיל את הקוד שנועד לטפל באירועים של הפקדים. בניגוד לפקדי Web Forms, מיקום ברירת המחדל של הפעלת הקוד עבור פקדי HTML הוא הדפדפן, ולא השרת - בהנחה שהדפדפן של המשתמש תומך באפשרות זו.

יש להוסיף לדף EmployeeForm.aspx מספר שיטות לטיפול באירועים: קבוצה של מטפלים שמטרתם למלא את הרשימה הנפתחת **PositionRole** בתפקידים כאשר המשתמש בוחר במשרה כלשהי (Worker, Boss, Vice President, President); מטפל סיום שמטרתו לשמור את הנתונים כאשר המשתמש לוחץ Save; ומטפל שמטרתו לאפס את הטופס כאשר המשתמש לוחץ Clear. בתרגיל הבא עליך לכתוב את המטפלים הללו.

כתוב מטפלים לאירועים

1. ב- Solution Explorer, הרחב את הקובץ EmployeeForm.aspx.

כתוצאה יופיע הקובץ EmployeeForm.aspx. בקובץ זה יאוחסן הקוד #C של המטפלים שתכתוב בהמשך. קובץ זה מכונה **code-behind file**. אלמנט זה של ASP.NET מאפשר להפריד בין קוד #C לבין הלוגיקה של התצוגה ביישום הרשת (למעשה ניתן לכתוב את קוד #C ואת המטפלים באירועים בקובץ EmployeeForm.aspx באמצעות חלון תצוגת הקוד, אבל גישה זו אינה מומלצת).

2. הצג את קוד HTML של קובץ EmployeeForm.aspx בחלון תצוגת הקוד. הבט בשורה הראשונה של הקובץ. השורה נראית כך:

```
<%@ Page Language="C#" ... CodeFile="EmployeeForm.aspx.cs ... %>
```

הנחיה זו מגדירה את תצורת הקובץ שמכיל את קוד התוכנית של טופס Web ואת שפת התכנות שבה נכתב, במקרה זה - #C. הטופס תומך גם בשפות Visual Basic ו-JScript.

3. ב- Solution Explorer לחץ לחיצה כפולה על הקובץ EmployeeForm.aspx.cs.

הקובץ יופיע בחלון Code and Text Editor, ובראשו תוצג סדרת משפטי **using**. שים לב שקובץ זה משתמש במרחב השמות **System.Web** ובתת-המרחבים שלו - שבהם נמצאות המחלקות השונות של ASP.NET. שים לב שהקוד עצמו נמצא במחלקה **_Default** אשר נגזרת מהמחלקה **System.Web.UI.Page**, שנגזרים ממנה כל טפסי Web. נכון לעכשיו, היא מכילה שיטה ריקה אחת בשם **Page_Load**, אשר מופעלת כאשר הדף נפתח. בשיטה זו ניתן לכתוב את הקוד הדרוש לאתחול הנתונים המשמשים את הטופס.

4. הוסף למחלקה **_Default** את השיטה **initPositionRole**, מיד לאחר השיטה **Page_Load**:

```
private void initPositionRole()
{
}
```

שיטה זו תשמש לאתחול הרשימה הנפתחת **positionRole** לערכי ברירת המחדל שלה.

5. הוסף את המשפטים הבאים לשיטה **initPositionRole**:

```
positionRole.Items.Clear();
positionRole.Enabled = true;
positionRole.Items.Add("Analyst");
positionRole.Items.Add("Designer");
positionRole.Items.Add("Developer");
```

המשפט הראשון מאפס את הרשימה. המשפט השני מפעיל (enable) את הרשימה (מיד תכתוב את הקוד אשר מכבה (disable) את הרשימה בנסיבות שונות). שאר המשפטים מוסיפים לרשימה שלושה תפקידים המתאימים למשרת עובד (Worker).

6. הוסף את המשפטים הבאים לשיטה Page_Load:

```
if (!IsPostBack)
{
   initPositionRole();
}
```

בלוק קוד זה יגרום לרשימה הנפתחת positionRole להתמלא כאשר הטופס נפתח בדפדפן של המשתמש. אולם חשוב להבין, שהשיטה **Page_Load** מופעלת בכל פעם שהשרת שולח את הטופס לדפדפן של המשתמש, ולא רק בפעם הראשונה. לדוגמה, כאשר המשתמש לוחץ על לחצן, הטופס עשוי להישלח בחזרה לשרת הרשת לעיבוד. תגובת השרת תהיה שליחת הטופס חזרה לדפדפן בגמר העיבוד. ייתכן מאוד שלא תרצה לבצע אתחול בכל פעם שדף האינטרנט מופיע, כי זהו בזבוז של משאבי עיבוד ועשוי לפגוע בביצועים בעת בניית אתרים מסחריים. ניתן להפעיל את השיטה **Page_Load** רק בהצגה הראשונה של הדף על ידי חקירת המאפיין **IsPostBack** של דף האינטרנט. מאפיין זה מחזיר ערך **false** בפעם הראשונה שהטופס מוצג, וערך **true** כאשר הדף מוצג שוב לאחר לחיצה על אחד הפקדים. בקוד שזה עתה הוספת, הקריאה לשיטה **initPositionRole** מתבצעת רק כאשר הטופס מוצג בפעם הראשונה.

7. חזור לקובץ EmployeeForm.aspx ועבור לתצוגת העיצוב. בחר בלחצן האפשרויות **workerButton**. בחלון Properties לחץ Events. לחץ לחיצה כפולה על האירוע **CheckedChanged**. אירוע זה מתרחש כאשר המשתמש לוחץ על לחצן האפשרויות וערכו משתנה. סביבת הפיתוח מחוללת את השיטה **workerButton_CheckedChanged** כדי שתטפל באירוע זה.

8. הוסף לשיטה **workerButton_CheckedChanged** את המשפט הבא:

```
initPositionRole()
```

זכור שערכי ברירת המחדל של הרשימה הנפתחת **positionRole** הם התפקידים המתאימים למשרת העובד, ולכן ניתן לעשות שימוש חוזר בשיטה initPositionRole.

9. חזור לקובץ EmployeeForm.aspx בתצוגת העיצוב. בחר בלחצן האפשרויות **bossButton**. השתמש בחלון Properties כדי ליצור את השיטה **bossButton_CheckedChanged** שתפקידה לטפל באירוע **CheckedChanged**. לאחר שהטופס יופיע בחלון Code and Text Editor, הוסף לשיטת האירוע את המשפטים הבאים:

```
positionRole.Items.Clear();
positionRole.Enabled = true;
positionRole.Items.Add("General Manager");
positionRole.Items.Add("Project Manager");
```

תפקידים אלה מתאימים למשרת המנהל.

10. חזור לקובץ EmployeeForm.aspx בתצוגת העיצוב וצור שיטה לטיפול באירוע **CheckedChanged** של לחצן האפשרויות **vpButton** (סגן מנהל). הוסף לשיטת האירוע את המשפטים הבאים:

```
positionRole.Items.Clear();
positionRole.Enabled = true;
positionRole.Items.Add("VP Sales");
positionRole.Items.Add("VP Marketing");
positionRole.Items.Add("VP Production");
positionRole.Items.Add("VP Human Resources");
```

11. השיטה האחרונה שתיצור צריכה לטפל באירוע **CheckedChanged** של לחצן האפשרויות President (נשיא). הוסף לשיטת האירוע את הקוד הבא:

```
positionRole.Items.Clear();
positionRole.Enabled = false;
```

חוקי התאמת התפקידים אינם תקפים לגבי מנהל המפעל, ולכן יש לכבות (disable) את הרשימה הנפתחת.

12. צור שיטה לטיפול באירוע Click של הלחצן Save. ביישום רגיל שיטה זו הייתה משמשת לשמירת נתונים בבסיס נתונים, אולם כדי לשמור על פשטות הדוגמה, השיטה רק תציג חלק מהנתונים שהמשתמש הזין באמצעות הפקד **InfoLabel**. הוסף את המשפטים הבאים לשיטה **saveButton_Click**:

```
string position = "";

if (workerButton.Checked)
   position = "Worker";
if (bossButton.Checked)
   position = "Manager";
if (vpButton.Checked)
   position = "Vice President";
if (presidentButton.Checked)
   position = "President";

infoLabel.Text = "Employee: " + firstName.Text + " " +
   lastName.Text + "   &nbspId " +
   employeeID.Text + "     Position " +
   position;
```

בשפת HTML, התו   מסמל רווח. בדרך כלל הדפדפן מתעלם מכל תווי הרווח הלבנים, מלבד הראשון שביניהם.

13. צור שיטה לטיפול באירוע Click של הלחצן Clear. הוסף לגוף השיטה את בלוק הקוד הבא:

```
firstName.Text = "";
lastName.Text = "";
employeeID.Text = "";
workerButton.Checked = true;
initPositionRole();
infoLabel.Text = "";
```

קוד זה נועד למחוק את הנתונים שהמשתמש הזין, ולאתחל את רשימת התפקידים ל-Worker (ערך ברירת המחדל).

בדיקה חוזרת של טופס Web

1. הפעל שוב את הטופס.
2. הקלד שם עובד וקוד עובד פיקטיביים. לחץ על הרשימה הנפתחת Role.
   הרשימה תציג את התפקידים המתאימים למשרת עובד.
3. שנה את המשרה של העובד הפיקטיבי לסגן מנהל (Vice President), ואחר כך לחץ שוב על הרשימה הנפתחת Role.
   שים לב שהרשימה טרם השתנתה ועודנה מציגה את התפקידים המתאימים למשרת העובד. הרשימה לא השתנתה, מכיוון שהאירוע **CheckedChanged** של לחצן האפשרויות Vice President לא התרחש.
4. סגור את חלון Internet Explorer וחזור לסביבת הפיתוח.
5. הצג את הקובץ EmployeeForm.aspx בחלון תצוגת העיצוב ובחר בלחצן האפשרויות **workerButton**. בחלון Properties שנה את ערך המאפיין **AutoPostBack** ל-True.
   כאשר המשתמש לוחץ על לחצן האפשרויות הזה, הטופס יישלח אל השרת לעיבוד, האירוע **CheckedChanged** יוצף, והטופס יעודכן כדי שיציג את התפקידים המתאימים ללחצן אפשרויות זה. על פי ברירת המחדל, ערך המאפיין AutoPostBack הוא False כדי למנוע תנועה לא נחוצה ברשת.
   שנה את ערך המאפיין **AutoPostBack** גם בלחצני האפשרויות האחרים: **bossButton**, **vpButton** ו-**presidentButton**.

טיפ

כדי לשנות את ערכי המאפיינים של כל ארבעת הלחצנים בבת אחת, סמן את כולם בעודך מחזיק את מקש Shift, ואחר כך שנה את המאפיינים הרלוונטיים בחלון Properties.

6. הפעל שוב את טופס Web. כאשר תלחץ הפעם על אחד מלחצני האפשרויות, המסך יהבהב לשנייה בשעה שהטופס נשלח לשרת, המטפל באירוע יופעל, הרשימה הנפתחת תעודכן והטופס יוצג שוב.
7. בתפריט View של Internet Explorer, לחץ Source כדי להציג את המקור של הדף HTML שמוצג בדפדפן. היישום Notepad יופעל ויציג את המקור HTML של דף האינטרנט. שים לב שבקובץ לא מופיעים פקדי Server מסוג "asp:" וגם לא קוד #C. הסיבה לכך היא שפקדי השרת על תכולתם הומרו לפקדי HTML (וחלקם גם ל-JavaScript). זו אחת התכונות הבסיסיות של פקדי Server - ניתן לגשת אליהם על ידי תכנות של שיטות, מאפיינים ואירועים, בדומה לאובייקטים רגילים של NET Framework. עם זאת, כאשר הם עוברים מהשרת לדפדפן, הם מומרים לקוד HTML. לאחר שתסיים לעיין בקובץ, סגור את Notepad.

8. לחץ Save.

הפקד **InfoLabel** יציג את נתוני העובד החדש. אם תתבונן במקור, תראה שקוד HTML עבור הפקד **InfoLabel** (אשר הפך לאובייקט span עם המזהה "InfoLabel") מכיל את אותו הטקסט.

9. לחץ Clear.

הטופס יציג שוב את ערכי ברירת המחדל שלו.

10. סגור את חלון Internet Explorer וחזור לסביבת הפיתוח.

עיבוד אירועים מרחוק

פקדי צד שרת הם ללא ספק כלי חזק של ASP.NET, אבל יש להם גם חסרונות. למרות שאירועים מוצפים על ידי הלקוח, הקוד לטיפול באירוע מופעל על ידי השרת וכל פעם שאירוע כלשהו מוצף, נשלחת דרך הרשת בקשת HTTP אל השרת. כעת, השרת מעבד את הבקשה ושולח תגובה המכילה דף HTML שיש להציג. בחלק גדול מהאירועים הדף החדש יהיה זהה לדף הקודם. השרת חייב לקבל גם את הנתונים האחרים שהמשתמש הזין, כי כאשר הוא מחולל דף אינטרנט חדש הוא צריך להציג בו גם אותם. אם שרת הרשת רק יחזיר את קוד HTML אשר יצר את הדף המקורי, כל הנתונים שהמשתמש הקליד לא יהיו כלולים בו. אם תתבונן במקור HTML של דף שנוצר באמצעות טופס Web, תבחין בשדה קלט מוסתר בו. הדוגמה שהוצגה קודם לכן כללה את השדה המוסתר הבא:

```
<input type="hidden" name="__VIEWSTATE"
value="/WEPdDwxNDk0MzA1NzE0O3Q8O2w8aTwxPjs+O2w8bDxpPDE3PjtpPDE5
PjtpP DIxPjtpPDI3PjtpPDMzPjs+O2w8dDxwPHA8bDxDaGVja2VkOz47bDxvPH
Q+Oz4+Oz 47Oz47dDxwPHA8bDxDaGVja2VkOz47bDxvPGY+Oz4+Oz47Oz47dDxw
PHA8bDxDaGVja2 VkOz47bDxvPGY+Oz4+Oz47Oz47dDx0PDt0PGk8Mz47QDxBbm
FseXN0O0Rlc2lnbmVyO0 RldmVsb3Blcjs+O0A8QW5hbHlzdDtEZXNpZ25lcjtE
ZXZlbG9wZXI7Pj47Pjs7Pj t0PHA8cDxsPFRleHQ7PjtsPFxlOz4+Oz47Oz47Pj
47Pj47bDxQZW9uQnV0dG9uO1BIQ kJ1dHRvbjtQSEJCdXR0b247VlBCdXR0b247
VlBCdXR0b247UHJlc2lkZW50QnV0dG9uO 1ByZXNpZGVudEJ1dHRvbjs+Pg==" />
```

מידע זה הוא למעשה תוכן הפקדים, או מצב התצוגה, בתצורה מקודדת. שדה זה נשלח לשרת הרשת בכל פעם שאירוע כלשהו גורם לשליחת בקשה. השרת משתמש במידע זה כדי למלא מחדש את השדות שבדף כאשר הוא מחולל את קוד HTML של התגובה. כל הפעולות האלו משפיעות על הביצועים. ככל שיש יותר פקדים על הטופס, כמות המידע שיש להעביר בין הדפדפן והשרת כחלק מהבקשות והתגובות עולה, וככל שתשתמש ביותר אירועים, חילופי המידע יתבצעו לעיתים תכופות יותר.

ככלל, כדי לצמצם את העברת המידע ברשת עליך להשתדל לבנות טפסי Web פשוטים יחסית; להימנע משימוש במספר גבוה מדי של אירועים; ולבחור במצבי תצוגה רק עבור פקדים הזקוקים לכך. כדי לבטל את מצב התצוגה של פקד, עליך לשנות את המאפיין EnableViewState שלו ל-False (ברירת המחדל היא True).

יצירת Theme ושימוש בו

כאשר יצרת אתר אינטרנט בפעם הראשונה, הגדרת סגנון עבור הטופס. הסגנון קבע את גופן ברירת המחדל וצבע ברירת המחדל של הפקדים בטופס. ניתן להשתמש בסגנון גם כדי לקבוע תכונות אחרות, כגון אופן הסידור והמספור של רשימות. (בתיבת הדו-שיח Style Builder בחר בכרטיסייה Lists כדי להציג את האפשרויות השונות לסידור רשימות.) סגנון שמוגדר באופן כזה ניתן להחיל על טופס אחד בלבד. אתרי אינטרנט מסחריים כוללים בדרך כלל עשרות, אם לא מאות טפסים. ניסיון לשמור על האחידות הסגנונית של הטפסים השונים עשוי לגזול זמן רב. (תוכל לתאר מצב שהחברה המעסיקה אותך מחליטה לשנות את הגופן של אתרי האינטרנט שלה, כמה טפסים יהיה עליך לעדכן ולבנות מחדש?) זו הסיבה שבגללה כדאי להשתמש ב-Themes, או **ערכות נושאים**. Theme הוא למעשה מקבץ של מאפיינים, סגנונות ותמונות שניתן להחיל על הפקדים של דף מסוים, או על כל הדפים של אתר כלשהו.

הערה

אם עבדת בעבר עם קבצי CSS (Cascading Style Sheets), הרעיון של ערכות נושאים בוודאי נשמע לך מוכר. אולם, יש מספר הבדלים בין השניים, ובעיקר: המדרג של ערכות הנושאים אינו דומה לזה של קבצי CSS. ובנוסף, מאפיינים שהוגדרו בערכת נושא ומוחלים על פקד, לעולם ידרסו את המאפיינים המקומיים שהוגדרו עבור אותו פקד.

הגדרת ערכות נושאים

ערכת נושא (Theme) מורכבת ממקבץ של קבצי skin המאוחסנים בתת-תיקייה של התיקייה App_Themes המשרתת אתר אינטרנט. קובץ skin הוא קובץ טקסט עם סיומת "skin". כל קובץ כזה מכיל את מאפייני ברירת המחדל עבור סוג פקד מסוים, בתור תחביר הדומה מאוד לזה שמופיע בעת הצגת טופס Web בחלון תצוגת המקור. לדוגמה, קובץ skin שלהלן מכיל את מאפייני ברירת המחדל עבור הפקדים TextBox ו-Label:

```
<asp:TextBox BackColor="Blue" ForeColor="White" Runat="Server" />
<asp:Label BackColor="White" ForeColor="Blue" Runat="Server" Font-
Bold="True" />
```

ניתן לשנות חלק גדול ממאפייני הפקד בקובץ skin, אך לא את כולם. לדוגמה, אינך יכול להגדיר את ערך המאפיין AutoPostBack. גם לא ניתן ליצור קבצי skin עבור כל סוגי הפקדים, אולם ניתן לקבוע את תצורתם של מרבית הפקדים הנמצאים בשימוש סדיר.

החלת ערכת נושא

לאחר יצירת מספר קבצי skin עבור Theme, תוכל להחיל אותו על דף אינטרנט על ידי שינוי התכונה @Page שמופיעה בראש הדף כאשר הוא מוצג בחלון תצוגת המקור. לדוגמה, אם קבצי skin של ערכת הנושא נמצאים בתיקייה App_Themes\BlueTheme של האתר, תוכל להחיל את הנושא על דף האינטרנט באופן הבא:

```
<%@Page Theme="BlueTheme" ...%>
```

כדי להחיל את הנושא על כל הדפים שבאתר, עליך לערוך את הקובץ Web.config ולציין את הנושא באלמנט של הדפים באופן הבא:

```
<configuration>
   <system.web>
      <pages theme="BlueTheme" />
   </system.web>
</configuration>
```

אם תשנה את הגדרות ה-Theme, כל הפקדים והדפים שערכת הנושא הוחלה עליהם יתעדכנו באופן אוטומטי בפעם הבאה שיוצגו.

במקבץ התרגילים האחרון לפרק זה, עליך ליצור ערכת נושא עבור האתר HonestJohn ולהחיל אותה על כל הדפים באתר.

יצירת ערכת נושא חדשה

1. בסביבת הפיתוח פתח את האתר HonestJohn אם אינו פתוח כבר.
2. ב- Solution Explorer, לחץ לחיצה ימנית על תיקיית הפרויקט HonestJohn. בתפריט הקיצור לחץ על Add ASP.NET Folder ולחץ Theme.

   תיקייה חדשה App_Themes תיווסף לפרויקט, ותיווצר גם תת-תיקייה בשם Theme1.
3. שנה את שם התיקייה Theme1 ל-**HJTheme**.
4. ב- Solution Explore, לחץ לחיצה ימנית על התיקייה HJTheme ובתפריט הקיצור בחר Add New Item.

   תיבת הדו-שיח Add New Item אשר תופיע, תציג את כל סוגי הקבצים שניתן לאחסן בתיקיית ערכות הנושאים.
5. בחר בתבנית Skin File, וקבע לקובץ את השם **HJ.skin**. לחץ Add.

   קובץ skin בשם HJ.skin ייווצר בתיקייה HJTheme, ותוכן הקובץ יוצג בחלון Code and Text Editor.

6. בחלון Code and Text Editor, הוסף את שורות הקוד הבאות לסוף הקובץ HJ.skin (הקובץ מכיל הערה הכוללת הוראות אחדות בקצרה):

```
<asp:TextBox BackColor="Red" ForeColor="White" Runat="Server" />
<asp:Label BackColor="White" ForeColor="Red" Runat="Server"
  Font-Bold="True" />
<asp:RadioButton BackColor="White" ForeColor="Red"
  Runat="Server"/>
<asp:Button BackColor="Red" ForeColor="White" Runat="Server"
  Font-Bold="True"/>
<asp:DropDownList BackColor="Red" ForeColor="White"
  Runat="Server"/>
```

מקבץ המאפיינים הפשוט שלעיל יגרום להצגת הפקדים **TextBox**, **button** ו-**DropDownListBox** בכיתוב לבן על רקע אדום, ושל הפקדים **Label** ו-**RadioButton** בכיתוב אדום על רקע לבן. הטקסט של המאפיין Label ו-Button יופיע בגרסה מודגשת של הגופן הנוכחי.

חשוב

עורך קבצי skin הוא בסיסי ביותר, ואינו כולל תפריטי Intellisense שיכולים לסייע לך. היישום יפעל גם אם תקליד נתונים מוטעים בקובץ זה, אבל הוא עשוי להתעלם מהרישומים המוטעים שבקובץ. כאשר תפעיל את היישום בהמשך, תצטרך לוודא שכל הפקדים מופיעים בו על פי הסגנון שהגדרת בערכת הנושא; אם לא, ודא שאין טעויות הקלדה בקובץ skin.

יש שתי דרכים לפחות להחלת ערכת נושא על טופס Web: ניתן לערוך את התכונה @Page שבכל אחד מהדפים, או לחילופין - לערוך את קובץ תצורת הרשת (Web configuration file) כדי להחיל את ערכת הנושא באופן גלובלי על האתר כולו. בתרגיל הבא נתרגל את הדרך השנייה מבין השתיים. שימוש במנגנון זה יגרום לכל הדפים החדשים שתוסיף לאתר זה ליישם באופן אוטומטי את אותה ערכה.

יצירת קובץ תצורת רשת והחלת ערכת הנושא

1. ב- Solution Explore לחץ לחיצה ימנית על הפרויקט Honest John ובתפריט הקיצור בחר Add New Item.

   תיבת הדו-שיח Add New Item אשר תופיע, תציג את כל סוגי הקבצים שניתן להוסיף לאתר האינטרנט.

2. בחר בתבנית Web Configuration File. ודא ששם הקובץ הוא Web.config ולחץ Add.

   הקובץ Web.config יתווסף לפרויקט ויוצג בחלון Code and Text Editor.

3. גלול לתחתית הקובץ Web.config, והוסף שורה מעל השורה **</system.web>**. הקלד את הרישום הבא בשורה החדשה:

```
<pages theme="HJTheme" />
```

4. בתפריט Debug בחר Start Without Debugging.

טיפ

אם חלון Internet Explorer מציג רשימת קבצים ולא את טופס Web, סגור את Internet Explorer וחזור לסביבת הפיתוח. ב- Solution Explorer, לחץ לחיצה ימנית על הקובץ EmployeeForm.aspx ובתפריט הקיצור בחר Set As Start Page. הפעל שוב את היישום.

5. דפדפן Internet Explorer יופיע ויציג את טופס Web. ודא שהסגנון של כל הפקדים בטופס השתנה בהתאם לנושא. הכיתוב של תיבות הטקסט עשוי להיות קצת קשה לקריאה, אך תתקן זאת מיד. סגור את Internet Explorer.

6. בסביבת הפיתוח הצג את קובץ HJ.skin בחלון Code and Text Editor. שנה את האלמנט המגדיר את מראה הפקדים **TextBox** ו-**DropDownList** באופן הבא:

```
<asp:TextBox BackColor="White" ForeColor="Red" Font-Bold="True"
Runat="Server" />
...
<asp:DropDownList BackColor="White" ForeColor="Red"
Runat="Server" />
```

7. הפעל שוב את הטופס. שים לב לשינוי בסגנון של **תיבות הטקסט** (First Name, Last Name ו-Employee Id) ושל **הרשימה הנפתחת** (Role). כעת הכיתוב בפקדים אלה ברור יותר. סגור את Internet Explorer.

☯ **אם ברצונך להמשיך לפרק הבא**

השאר את Visual Studio 2005 פעילה ועבור לפרק 26.

☯ **אם ברצונך לכבות כעת את Visual Studio 2005**

פתח את תפריט Menu ולחץ Exit. אם מופיעה תיבת דו-שיח Save, לחץ Yes.

פרק 25 – טבלה מסכמת

המשימה	צריך
ליצור יישום רשת.	צור אתר חדש בעזרת התבנית ASP.NET Web Site. ציין האם ברצונך להשתמש בשרת Development Web Server (פרט את שם הקובץ של קובץ המערכת ואת מיקומו), או בשרת IIS (פרט כתובת HTTP ו-URL).
לפתוח את ההגדרות HTML של טופס Web ולערוך אותן.	עבור לתצוגת המקור בחלון תצוגת העיצוב.
לבחור בסגנון עבור טופס Web.	בתפריט Format בחר Style. בתיבת הדו-שיח Style עצב את הסגנון הרצוי לטופס שלך.
להוסיף פקדי צד שרת לטופס Web.	עבור לתצוגת העיצוב בחלון תצוגת העיצוב. בערכת הכלים הרחב את הקטגוריה Standard. גרור פקדים אל טופס ה- Web.
להוסיף פקדי HTML לטופס Web.	בערכת הכלים הרחב את הקטגוריה HTML. גרור פקדים אל טופס ה-Web.
ליצור שיטה לטיפול באירוע של פקד Server.	בתצוגת העיצוב סמן את הפקד שעל הטופס. בחלון Properties לחץ Events. אתר את האירוע שאתה רוצה לטפל בו, והקלד את שם השיטה לטיפול באירוע. בחלון Code and Text Editor הוסף לשיטה החדשה את הקוד המתאים.
להציג את הקוד HTML עבור טופס ה-Web בסביבת הריצה.	בתפריט View של Internet Explorer, בחר Source. המקור HTML יוצג באמצעות היישום Notepad.
ליצור ערכת נושא (Theme).	הוסף לאתר האינטרנט תיקייה בשם App_Themes. הוסף תת-תיקייה עבור הנושא. צור קובץ skin המגדיר את מאפייני הפקדים שבטופס.

להחיל את הנושא על האתר.	החל את ערכת הנושא דרך התכונה @Page של כל דף, כלהלן:
	`<%@Page Theme="BlueTheme" ...%>`
	לחילופין, ערוך את הקובץ Web.config והוסף את ערכת הנושא לאלמנט של הדפים, כלהלן:
	`<configuration>`
	`  <system.web>`
	`  <pages theme="BlueTheme" />`
	`  </system.web>`
	`</configuration`

פרק 26

פקדי בדיקת תקינות עבור טפסי Web

כאשר תסיים פרק זה, תוכל:

- לבדוק את תקינות הקלט שמתקבל מהמשתמש בטופס Web של ASP.NET באמצעות **פקדי תקינות** (validation controls).
- לקבוע את מיקום ביצוע הבדיקות (בצד השרת או במסגרת הדפדפן).

בדומה ליישומי הטפסים, Windows Forms, בדיקת תקינות של הקלט מהמשתמש ביישומי Web Forms מהווה נדבך חשוב בפעולת המערכת. במקרה של טפסי Windows האפשרויות, העיתוי והמיקום של בדיקות התקינות היו מוגבלות - וידאת שהקלט אינו חסר משמעות באמצעות אירועים השייכים לפקדים ולטפסים של היישום עצמו. במקרה של טפסי Web, עליך להתחשב בגורם נוסף: האם עליך לבצע בדיקות אלו במסגרת השרת או במסגרת הדפדפן? בפרק זה תלמד מהם השיקולים בקבלת ההחלטה ומהן האפשרויות הניצבות בפניך.

השוואה בין בדיקות תקינות במסגרת השרת לבין מסגרת הלקוח

נחזור שוב לדף EmployeeForm.aspx שבאתר HonestJohn, שבו המשתמש מזין נתונים של עובד בחברה: שם, קוד עובד, משרה ותפקיד. חובה עליו למלא את כל תיבות הטקסט ולהקפיד שקוד העובד יהיה מספר שלם חיובי.

אם היה זה יישום Windows Forms, היית משתמש באירוע **Validating** כדי לוודא שהמשתמש הזין ערך כלשהו בתיבות הטקסט First Name ו-LastName ושקוד העובד הוא נומרי. אבל בטפסי Web אין אירוע **Validating**, ולכן עליך להשתמש בדרך בדיקה שונה.

בדיקות תקינות במסגרת השרת

אם תבחן מקרוב את המחלקה **TextBox** תראה שהיא מציפה אירוע **TextChanged**. אירוע זה מוצף כאשר הטופס נשלח חזרה לשרת לאחר שהמשתמש שינה את הטקסט שבתיבת הטקסט. בדומה לכל האירועים של פקדי Server, גם האירוע **TextChanged** מופעל בשרת. הפעולה כוללת העברת נתונים מהדפדפן לשרת, עיבוד האירוע במסגרת השרת כדי לבדוק את תקינות הנתונים, ולבסוף - לארוז שגיאות אפשריות (אם נתגלו באירוע) באריזת HTML כמענה שמוחזר ללקוח. אם בדיקות תקינות לנתונים מורכבת או מחייבת עיבוד שניתן לבצע רק במסגרת השרת (למשל, לוודא שקוד העובד מתאים לסוגים קיימים במסד נתונים), אז זו הדרך

שכדאי להשתמש בה. אבל, אם ברצונך לבדוק נתון יחיד בתיבת טקסט (למשל, לוודא שקוד העובד שהמשתמש הזין הוא מספר שלם חיובי), ביצוע הבדיקה במסגרת השרת יגרום לתנועה מיותרת של נתונים ברשת ולפגיעה בביצועים. ונשאלת השאלה, מדוע לא לבצע בדיקות אלו במחשב של הלקוח וכך לחסוך את כל העברות הנתונים המיותרות.

מתן תוקף במסגרת הלקוח

מודל Web Forms מאפשר בדיקות תקינות במסגרת הלקוח על ידי שימוש בפקדי תקינות נתונים (validation controls). אם המשתמש מציג את הדפים באמצעות דפדפן כגון Internet Explorer 4 או גרסה מאוחרת יותר, אשר תומך בדפי HTML דינמיים, פקדים אלה יחוללו קוד JavaScript שפועל במסגרת הדפדפן כדי למנוע את הצורך לשלוח את הנתונים לשרת ולהמתין לתגובה. במקרה של דפדפנים ישנים יותר, פקדי התוקף יחוללו את הקוד בצד השרת. יתרונם העיקרי של פקדים אלה בכך, שבעת בניית טופס Web אינך צריך לדאוג לדברים הללו. פקדי תקינות מכילים כלים מובנים המאפשרים להם לזהות את סוג הדפדפן ולחולל את הקוד המתאים. כמפתח היישום, כל שעליך לעשות זה לגרור את הפקדים אל הטופס, לקבוע את מאפייניהם (על ידי כתיבת קוד או בעזרת חלון Properties), ולהגדיר את כללי התקינות שעל הפקדים לאכוף, ואת הודעות השגיאה שעליהם להציג.

המודל של ASP.NET כולל חמישה פקדי תקינות:

- **RequiredFieldValidator** השתמש בפקד זה כדי לוודא שהמשתמש הזין נתון בפקד.
- **CompareValidator** השתמש בפקד זה כדי להשוות את הנתון שהוזן עם ערך קבוע, ערך המאפיין של פקד אחר או ערך שנשלף ממאגר נתונים.
- **RangeValidator** השתמש בפקד זה כדי להשוות את הנתון שהוזן כנגד תחום ערכים, כדי לבדוק האם הנתון משתייך לתחום הערכים הזה או לא.
- **RegularExpressionValidator** השתמש בפקד זה כדי לוודא שהנתון אותו הזין המשתמש תואם לביטוי, לתבנית או לפורמט מוגדר (כגון מספר טלפון).

הערה

אין הגבלה על הטקסט שהמשתמש יכול להזין לתיבת הטקסט ולשלוח לשרת. הוא יכול גם להקליד טקסט שנראה כמו תגיות HTML (<b> לדוגמה). האקרים משתמשים לעתים בטכניקה זו כדי להוסיף הוראות HTML לבקשת הלקוח, כדי לפגוע בשרת או כדי לפרוץ אליו. על פי ברירת המחדל, כל ניסיון שכזה בדפי אינטרנט של ASP.NET יגרום לביטול הבקשה ולמשתמש תישלח הודעה שתרגומה לעברית הוא "ערך Request.Form שעשוי להיות מסוכן התגלה אצל הלקוח". ניתן לבטל את הבדיקה, אבל אין זה מומלץ. גישה טובה יותר תהיה להשתמש בפקד **RegularExpressionValidator** כדי לוודא שהקלט שהתקבל מהמשתמש באמצעות תיבת הטקסט אינו תגית HTML, או כל דבר אחר שנראה כמו תגית. למידע נוסף על Regular Expressions ואופן השימוש בהם, ראה "NET Framework Regular Expressions." ב- Microsoft Visual Studio 2005 Documentation.

- **CustomValidator** השתמש בפקד זה כדי להגדיר בדיקה מותאמת משלך, ולשייך אותה לפקד לשם בדיקת תוקף.

למרות שכל אחד מהפקדים מבצע רק סוג אחד של בדיקת תוקף, ניתן לשלב כמה מהם יחדיו. לדוגמה, אם ברצונך לוודא שהמשתמש אכן מקליד ערך לתיבת הטקסט ושערך זה נכלל בתחום ערכים מסוים, עליך לשייך לתיבת הטקסט את הפקדים **RequiredFieldValidator** ו-**RangeValidator**. פקדים אלה, בשילוב הפקד **ValidationSummary**, יכולים לפעול יחדיו כדי להציג הודעות שגיאה. בתרגילים הבאים תתרגל שימוש בחלק מהפקדים הללו.

מימוש בדיקת תקינות במסגרת הלקוח

אם תחזור לטופס EmployeeForm.aspx, תוכל לראות שיש לצרף פקדי **RequiredFieldValidator** לתיבות הטקסט First Name, Last Name ו- Employee Id. בנוסף, קוד העובד חייב להיות מספר שלם חיובי. עבור היישום הזה עליך להגדיר שערך קוד העובד חייב להיות בין 1 ל-5000. עשה זאת על ידי הוספת הפקד **RangeValidator**.

הוסף פקדי RequiredFieldValidator

1. בסביבת התכנות של Visual Studio 2005, פתח את תפריט File, Open ובחר Web Site. בתיבת הדו-שיח Open Web Site, ודא שהאפשרות File System מסומנת ופתח את התיקייה My Document\Microsoft Press\Visual CSharp Step by Step\Chapter 26\HonestJohn. לחץ Open.

הערה

אינך צריך לבחור בפתרון או בקובץ פרויקט של #C כדי לפתוח אתר אינטרנט למטרות עריכה. עליך לפתוח את התיקייה המכילה את קבצי האתר ואת תת-התיקיות. סביבת הפיתוח תחולל עבורך קובץ פתרון חדש אם התיקייה שבחרת אינה מכילה קובץ שכזה, ותצרף אליו את קבצי האתר. האתר עצמו אינו זקוק לקובץ פתרון זה, ולכן אינך צריך לשמור אותו.

הערה

בעת יצירת אתר אינטרנט חדש, סביבת הפיתוח יוצרת קובץ פתרון בתוך תיקיית הפתרון שבתיקייה My Documents\Visual Studio 2005. אם ברצונך לפתוח את האתר באמצעות קובץ פתרון קיים ולא ליצור קובץ חדש, עליך לבחור בתפריט File, Open את Project/Solution. פתח את תיקיית הפתרון ולחץ על שם הקובץ.

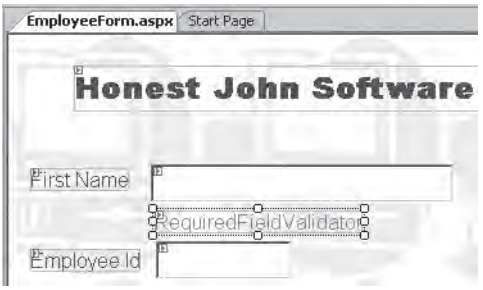
2. ב- Solution Explorer, לחץ לחיצה ימנית על EmployeeForm.aspx, ובתפריט הקיצור בחר Set As Start Page.

3. לחץ שוב לחיצה ימנית על הקובץ EmployeeForm.aspx, ובתפריט הקיצור בחר הפעם View Designer כדי להציג את הטופס Web בחלון תצוגת העיצוב.

4. בערכת הכלים הרחב את הקטגוריה Validation.

5. הוסף לטופס את הפקד RequiredFieldValidator.

הפקד יופיע בחלקו הימני עליון של הטופס, ועליו יוצג הכיתוב "RequiredFieldValidator". גרור את הפקד אל תחת תיבת הטקסט First Name בדומה למסך הבא. ייתכן שתצטרך להפעיל את האפשרות absolute positioning של הטופס (בתפריט Layout בחר Position) לפני שתוכל להזיז את הפקד RequiredFieldValidator.



6. כאשר **RequiredFieldValidator** מסומן, לחץ על חלון Properties. שנה את המאפיין **ControlToValidate** ל-**firstName**. במאפיין **ErrorMessage** הקלד "**You must type a first name for the employee**".

   זוהי ההודעה המופיעה כאשר לא מוזן נתון לפקד שבו מתבצעת הבדיקה (תיבת הטקסט First Name).

7. הוסף לטופס שני פקדי **RequiredFieldValidator** נוספים. מקם את הפקד הראשון מתחת לתיבת הטקסט Last Name, שנה את המאפיין **ControlToValidate** ל-lastName, ובמאפיין **ErrorMessage** הקלד "**You must type a last name for the employee**". שנה את גודל הפקד כדי שהודעת השגיאה תופיע בשורה אחת בלבד.

   מקם את הפקד השני מתחת לתיבת הטקסט Employee ID, שנה את המאפיין **ControlToValidate** ל-employeeID ובמאפיין **ErrorMessage** הקלד "**You must specify an employee ID**".

8. בתפריט Debug בחר Start Without Debugging כדי לבנות ולהפעיל את הטופס.

9. כאשר הטופס ייפתח בחלון Internet Explorer, כל תיבות הטקסט יהיו ריקות. לחץ Save. הודעת השגיאה אשר שייכת לשלושת פקדי **RequiredFieldValidator** תופיע על המסך.

   שים לב שהאירוע **Click** של הלחצן Save לא הוצף ושהתווית שבתחתית הטופס אינה מציגה את סיכום הנתונים, וגם המסך אינו מהבהב. הסיבה לכך היא שפקדי התוקף מנעו את שליחת הבקשה לשרת. הם חוללו קוד שהדפדפן יכול להציג, וימשיכו למנוע את שליחת הנתונים עד אשר תתקן את כל השגיאות.

10. הזן שם לתיבת הטקסט First Name.

    ברגע שתעבור מהתיבה הזו, הודעת השגיאה תיעלם. אם תחזור לתיבת הטקסט First Name, תמחק את תוכן התיבה ותעבור לתיבת הטקסט הבאה, הודעת השגיאה תופיע שוב. כל התהליכים הללו מתבצעים במסגרת הדפדפן, מבלי לשלוח דבר אל השרת.

11. הזן ערכים לתיבות הטקסט First Name, Last Name ו- Employee ID. ואחר כך לחץ Save. האירוע **Click** יופעל הפעם, וסיכום הנתונים שהזנת יופיע בפקד InfoLabel שבתחתית הטופס.

12. סגור את הטופס וחזור לסביבת הפיתוח.

במצב הנוכחי של היישום ניתן להזין כל סוג של נתון אל תיבת הטקסט Employee ID. בתרגיל הבא עליך להשתמש בפקד **RangeValidator**, כדי להגביל את הערכים שהמשתמש יכול להזין למספרים שלמים בין 1 ל-5000.

הוסף פקד RangeValidator

1. בערכת הכלים, הוסף פקד **RangeValidator** לטופס וגרור אותו אל מתחת לפקד **RequiredFieldValidator** אשר נמצא מתחת לתיבת הטקסט Employee ID. זכור, ייתכן שתצטרך להזיז את התווית Position ולחצני האפשרויות כדי לפנות מקום לפקד החדש.

2. כאשר **RangeValidator** מסומן, לחץ על חלון Properties. שנה את המאפיין **ControlToValidate** ל-employeeID. במאפיין **ErrorMessage** הקלד "**ID Must be between 1 and 5000**". במאפיין **Maximum Value** הצב את הערך 5000, ובמאפיין **Minimum Value** הצב 1. במאפיין **Type** בחר Integer.

3. הפעל שוב את הטופס. הקלד שם פרטי ושם משפחה, אבל את קוד העובד השאר ריק. לחץ Save.

   הודעת השגיאה תציין שעליך להזין קוד עובד.

4. הקלד 1- בתיבת הטקסט Employee Id ולחץ Save.

   גם הפעם תופיע הודעת שגיאה, אך הפעם היא תציין שקוד העובד חייב להיות בין 1 ל-5000.

5. הקלד **101** בתיבת הטקסט Employee Id ולחץ Save. הפעם הבדיקה תעבור בהצלחה. הטופס יישלח חזרה לשרת, האירוע **Click** של הלחצן Save יופעל, וסיכום נתוני העובד שהזנת לטופס יופיע בתווית **InfoLabel** שבתחתית הטופס.

6. נסה להקליד ערכים אחרים שאינם בתחום, או שאינם מהסוג הנכון. נסה להזין 5001 ואת המחרוזת "AAA" כדי לבדוק אם הפקד **RangeValidator** פועל כמצופה.

7. בתפריט View של Internet Explorer, בחר **Source**.

   הקוד HTML של האתר יופיע ביישום Notepad. עיין בטופס ובחן את תוכנו. בחלק התחתון של הטופס תבחין בבלוק קוד בשפת JavaScript אשר נועד לבצע את בדיקות התקינות. סביבת ההרצה מחוללת את הקוד הזה על פי ערכי המאפיינים של פקדי הבדיקה. לאחר שתסיים לעיין בקובץ, סגור את Notepad.

8. סגור את הטופס וחזור לסביבת הפיתוח Visual Studio 2005.

ביטול בדיקת התקינות אצל הלקוח

בתרגיל האחרון ראית כיצד בדיקת התקינות מבוצעת על ידי שימוש בקוד JavaScript שמופעל בדפדפן. סביבת ההרצה של APS.NET מחוללת את הקוד הזה באופן אוטומטי, על פי יכולות הדפדפן. אם הדפדפן אינו תומך בקוד JavaScript, כל תהליך בדיקת התקינות יבוצע על ידי קוד שמופעל בשרת.

ניתן לחסום את בדיקת התקינות במסגרת הלקוח ולהעביר את כל הבדיקות למסגרת השרת על ידי שינוי ערך המאפיין **EnableClientScript** ל-False. ייתכן שתצטרך לעשות זאת בנסיבות מסוימות, כמו למשל בעת בדיקות מורכבות במיוחד שהגדרת בעצמך באמצעות הפקד **CustomValidator**, או בבדיקות שמחייבות גישה למסדי נתונים שנמצאים בשרת. בנוסף, הפקד **CustomValidator** כולל אירוע **ServerValidate** שמאפשר לבצע את בדיקת התקינות במסגרת השרת גם כאשר ערך המאפיין **EnableClientScript** הוא True.

ראית כיצד הפקדים משמשים לבדיקת הנתונים המתקבלים מהמשתמש, הבעיה היא שתצוגת הודעת השגיאה איננה נאה במיוחד. בתרגיל הבא עליך להשתמש בפקד **ValidationSummary** כדי לשנות את אופן הצגת הודעות השגיאה בפני המשתמש.

הוסף פקד ValidationSummary

1. בטופס EmployeeForm.aspx סמן את הפקד **RequiredFieldValidator1** שנמצא תחת תיבת הטקסט First Name. בחלון Properties שנה את המאפיין Text לכוכבית (*).

   ההודעה You must enter a first name for the employee תוחלף בכוכבית (*) כי הפקדים מציגים את המאפיין **Text** על הטופס. אם לא תבחר ערך עבור המאפיין **Text**, הוא יקבל את הערך של המאפיין **ErrorMessage**.

2. הצב מחדש את הפקד **RequiredFieldValidator1** מימין לתיבת הטקסט First Name. במקרה של שגיאה תופיע כוכבית אדומה ליד תיבת הטקסט שהיא המקור לשגיאה.

3. סמן את הפקד **RequiredFieldValidator2**, שנה את המאפיין Text שלו לכוכבית (*), והצב אותו מימין לתיבת הטקסט Last Name.

4. סמן את הפקד **RequiredFieldValidator2**, שנה את המאפיין **Text** שלו לכוכבית (*), והצב אותו מימין לתיבת הטקסט Employee Id. פעל כך גם עבור הפקד **RangeValidator1**.

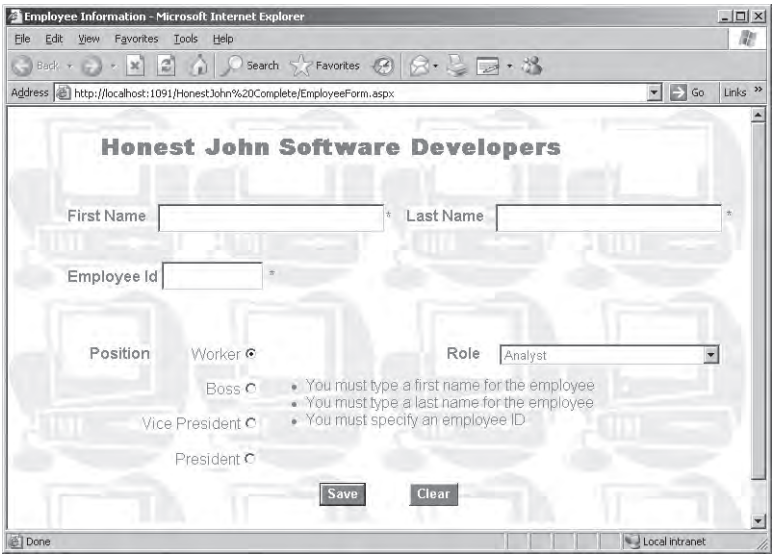
5. בערכת הכלים, הוסף לטופס פקד **ValidationSummary** והצב אותו במרווח שבין הלחצנים לבין לחצני האפשרויות.

   כתוצאה, יופיע פקד **ValidationSummary** אשר מציג את ערכי **ErrorMessage** של כל פקדי התוקף שעל הטופס.

6. אל תשנה את גודל הפקד **ValidationSummary**. ודא שערך המאפיין **ShowSummary** שלו הוא True.

7. הפעל את טופס Web. בטופס שיופיע, השאר את תיבות הטקסט First Name, Last Name ו–Employee ID ריקות, ולחץ Save.

כוכביות אדומות יופיעו לצד כל אחת מתיבות הטקסט, והודעות השגיאה המתאימות יופיעו בפקד **ValidationSummery** שבתחתית הטופס.



8. הקלד שם פרטי ושם משפחה בתיבות הטקסט המתאימות, ואת המחרוזת **AAA** בתיבת הטקסט Employee Id.

כאשר תעבור מתיבה לתיבה, ייעלמו הכוכביות מתיבות הטקסט First Name ו– Last Name, אולם כוכבית נוספת תופיע ליד תיבת הטקסט Employee Id.

9. לחץ Save.

הודעת השגיאה שבפקד **ValidationSummary** תשתנה.

10. בתיבת הטקסט Employee Id הקלד **101**, ולחץ Save.

כעת ייעלמו כל הכוכביות והודעות השגיאה, וסיכום הנתונים שהתקבלו יופיע בפקד **InfoLabel**.

11. סגור את הטופס וחזור לסביבת הפיתוח Visual Studio 2005.

דפי HTML דינמיים והודעות שגיאה

אם ברשותך דפדפן שתומך בדפי HTML דינמיים, תוכל להציג את הודעת השגיאה בתיבת הודעה ולא על פני הטופס. כדי לעשות זאת, שנה את ערך המאפיין **ShowMessageBox** של הפקד **ValidationSummary** ל-True. בסביבת ההרצה, ובמקרה של שגיאה בתהליך הבדיקות, הודעת השגיאה תוצג בתיבת הודעה.

☯ **אם ברצונך להמשיך לפרק הבא**

השאר את Visual Studio 2005 פעילה ועבור לפרק 27.

☯ **אם ברצונך לכבות כעת את Visual Studio 2005**

פתח את תפריט Menu ולחץ Exit. אם מופיעה תיבת דו-שיח Save, לחץ Yes.

פרק 26 – טבלה מסכמת

המשימה	צריך
לבצע בשרת את בדיקת התקינות של הקלט מהמשתמש.	השתמש באירועים השייכים לפקדים עצמם, כמו לדוגמה, האירוע **TextChanged** של הפקד **TextBox**.
לבצע במחשב הלקוח את בדיקת תקינות הקלט מהמשתמש.	השתמש בפקדי בדיקות התקינות. במאפיין **ControlToValidate** בחר את הפקד שברצונך לבדוק, ובמאפיין **ErrorMessage** הצב את הודעת השגיאה שברצונך להציג. ודא שערך המאפיין **EnableClientScript** הוא True.
לחייב את המשתמש להציב ערך כלשהו בתיבת טקסט.	השתמש בפקד **RequiredFieldValidator**.
לבדוק את הסוג והתחום של ערכי הנתונים שהמשתמש הקליד בתיבות הטקסט.	השתמש בפקד **RangeValidator**. הצב את הערכים הרצויים במאפיינים **Type**, **MaximumValue** ו-**MinimumValue**.
להציג סיכום של הודעות השגיאה.	השתמש בפקד **ValidationSummary**. ודא שערך המאפיין **ShowSummery** הוא אכן True.

פרק 27

אבטחת אתרים וגישה לנתונים באמצעות טפסי Web

כאשר תסיים פרק זה, תוכל:

- לאבטח את הגישה לאתר באמצעות פקדי Login ו- Form-based authentication.
- ליצור טפסי Web אשר מציגים מידע ממסד נתונים באמצעות פקד GridView.
- לעדכן מסד נתונים באמצעות טופס Web.
- לבנות יישום שיכול להציג כמויות גדולות של נתונים, ובאותה העת גם למזער את השימוש במשאבים.

בפרק זה ניגע בעיקרי הדברים בלבד. כל מי שמעוניין להרחיב את ידיעותיו ראוי שיתחיל בקריאת הספר "**מדריך אבטחת מידע והגנה מפני האקרים**" מאת דורון סיון שיצא בהוד-עמי. בארץ פועלים מומחים בתחום האבטחה, כמו אליק לוין ממיקרוסופט, שכותבים מאמרים המתפרסמים מפעם לפעם בעתונות המקצועית, ולחלקם יש אתרים או בלוג.

בשני הפרקים האחרונים למדת לבנות אתר אינטרנט אשר מאפשר למשתמש להזין מידע ואף לבצע בדיקת תקינות למידע שמתקבל. בפרק זה תלמד ליצור יישומים אשר מציגים נתונים ממסד נתונים ומעדכנים אותו על פי השינויים שהמשתמש מגדיר, תוך מזעור השימוש במשאבים משותפים, כגון הרשת עצמה ומסד הנתונים.

אבטחה הייתה תמיד סוגייה מרכזית, בעיקר בעת בנייה של יישומים שהגישה אליהם מתאפשרת דרך האינטרנט. לכן, עליך גם ללמוד כיצד לשלב בטפסים שתבנה את מנגנוני האבטחה, שמטרתם לאמת את זהות המשתמש במסגרת הטופס.

שימוש בפקד GridView של Web Forms

כאשר למדת בפרקים הקודמים כיצד לגשת למסדי נתונים, למדת למעשה כיצד להשתמש בפקד DataGridView המיועד עבור Windows Forms. לטפסי Web יש פקד דומה בשם **GridView**, אשר שונה מהפקד **DataGridView**, מכיוון שמיקרוסופט התאימה אותו לעבודה בסביבת ASP.NET, אבל למעשה, הייעוד של שניהם דומה: להציג ולערוך שורות שנשלפו ממקור נתונים. אחד ההבדלים ביניהם קשור להעברה והצגה של כמויות נתונים גדולות. ביישומי Web Forms, הסבירות שיישום הלקוח (הדפדפן) יהיה מרוחק ממסד הנתונים היא גבוהה. לכן, במקרה שכזה חשוב מאוד לנצל בחוכמה את רוחב הפס (bandwidth), ולא לבזבז משאבים על שליפת כמויות גדולות של נתונים אשר המשתמש כלל אינו זקוק להם. הפקד **GridView** של Web Forms תומך במנגנון **הדפדוף** (paging) המאפשר לשלוף נתונים על פי דרישה, בשעה שהמשתמש גולל מעלה ומטה את הנתונים.

בדומה לפקד **DataGridView** של Windows Form, הפקד **GridView** של Web Forms יכול לפעול גם כאשר הוא מנותק ממסד הנתונים. תוכל ליצור אובייקט מסוג **SqlDataSource** כדי להתחבר למסד הנתונים, למלא את האובייקט **DataSet**, ואז להתנתק ממסד הנתונים. ניתן גם לכרוך את האובייקט **DataSet** שבפקד **SqlDataSource** לפקד **GridView**. בניגוד לפקד **DataGridView** של Windows Forms, הנתונים המוצגים בפקד **GridView** של Web Forms מוצגים בטבלה לקריאה בלבד (בתור טבלת HTML בדפדפן). מאפייני הפקד **GridView** מאפשרים למשתמש לעבור למצב עריכה, שבו הופכות השורות המסומנות למקבץ של תיבות טקסט ואשר באמצעותן הוא יכול לערוך את הנתונים. תוכל לתרגל טכניקה זו בתרגילים שבהמשך הפרק.

ניהול האבטחה

בעת בניית יישומים באמצעות .NET Framework, עומדים לרשותך מנגנונים רבים שיכולים לשמש לבדיקה אם למשתמשים המפעילים את היישום מותר לעשות זאת. חלק מטכניקות האימות הללו מסתמכות על קבלת שם מזהה וסיסמה, בעוד שטכניקות אחרות מסתמכות על מנגנוני אבטחה פנימיים של מערכת ההפעלה. ככל הנראה, לא תוכל להשתמש במנגנוני האבטחה הפנימיים של Windows בעת בנייה של יישומי רשת שהגישה אליהם תתבצע דרך האינטרנט - המשתמשים ביישום יהיו ככל הנראה חברים ב-domain של Windows שאינו מוכר ליישום, או שהם ישתמשו במערכת הפעלה אחרת כגון UNIX. לכן, האפשרות הטובה ביותר לאבטחת יישומי רשת היא **אבטחה במסגרת הטופס**.

אבטחה במסגרת הטופס

אבטחה במסגרת הטופס (Form-based authentication) מאפשרת לאמת את זהות המשתמש על ידי הצגת טופס login, שבו הוא יתבקש להזין שם משתמש וסיסמה. לאחר שזהות המשתמש מאומתת, הוא יוכל לגשת לטפסי Web המרכיבים את היישום, ובמקרה שיש צורך באימות נוסף, יהיה ניתן לעשות זאת באמצעות קוד בכל אחד מדפי האינטרנט האחרים (ייתכן שלמשתמש יש גישה למערכת, אבל אין לו גישה לכל חלקי היישום). כדי להשתמש באבטחה במסגרת הטופס של ASP.NET, עליך לשנות את תצורת היישום על ידי עריכת קובץ התצורה Web.config, ועליך גם ליצור טופס שנועד לאמת את זהות המשתמש. טופס האבטחה יוצג בכל פעם שהמשתמש ינסה לגשת לכל חלק של היישום לפני שזהותו אומתה. המשתמש יוכל להמשיך לדף הרצוי רק אם זהותו תאומת על פי ההיגיון שמאחורי טופס הכניסה.

חשוב

חסרי הניסיון שביניכם עשויים לחשוב שהמנגנונים של ASP.NET לאבטחה במסגרת הטופס מוגזמים במקצת, אבל אין זה נכון. אל תתפתה ליצור טופס כניסה המשמש כשער כניסה ליישום שלך מתוך נקודת הנחה שהמשתמש ייכנס תמיד לאתר דרך טופס זה. דפדפנים יכולים להטמין (cache) טפסים וכתובות URL באופן מקומי במחשב של המשתמש. משתמש אחר עשוי להשיג גישה למטמון של המחשב (יכולת זו תלויה באופן הגדרת המחשב עצמו), למצוא כתובות URL של חלקי יישום רגישים, ולפתוח אותם ישירות מבלי לעבור דרך טופס הכניסה. לך כמתכנת, יש שליטה על השרת, אולם כמעט ואין לך שליטה על המחשב של המשתמש. המנגנון של ASP.NET לאבטחה במסגרת הטופס הוא יציב למדי, ובהנחה שהשרת שלך מאובטח בצורה טובה, מנגנון זה מתאים לרוב היישומים שתבנה.

מימוש האבטחה במסגרת הטופס

בתרגילים הראשונים שבפרק זה עליך ליצור יישום ולהגדיר את תצורתו. בסוף התהליך, יישום זה יאפשר למשתמש לצפות בנתוני הלקוחות שבמסד הנתונים Northwind ולערוך אותם.

צור את אתר האינטרנט Northwind

1. בסביבת התכנות של Visual Studio 2005, צור אתר ASP.NET חדש בשם Northwind (בסעיף Location בחר File System). צור את האתר בתיקייה \My Documents\Microsoft Press\Visual CSharp Step by Step\Chapter 27\Northwind. קבע את השפה C# עבור האתר.
2. ב- Solution Explorer, שנה את שם הטופס Default.aspx ל-CustomerData.aspx.
3. לחץ לחיצה ימנית על CustomerData.aspx ובתפריט הקיצור בחר Set As Start Page.
4. בחלון תצוגת המקור שמוצג בו קוד HTML של הטופס, עבור לכרטיסייה Design.
5. בתפריט Layout, Position בחר Auto-position Options. בתיבת הדו-שיח Options, סמן את תיבת הסימון "Change positioning to the following for controls added using the Toolbox, paste, or drag and drop". בנוסף, עליך לוודא שברשימה הנפתחת הפריט "Absolutely positioned" אכן מסומן. לחץ OK.
6. הוסף פקד **Label** מערכת הכלים. גרור אותו למרכז הטופס **CustomerData**. במאפיין **Text** של התווית הקלד **This form will be implemented later**.

בתרגילים הבאים עליך לבנות טופס login לאימות זהות המשתמש ולהגדיר את תצורת האבטחה במסגרת הטופס של יישום הרשת. טופס הכניסה יוצג בכל פעם שמשתמש שזהותו לא אומתה, ינסה לקבל הרשאת גישה ליישום. לאחר שתגדיר שעליה להשתמש באבטחה במסגרת הטופס, סביבת ההרצה של ASP.NET תנתב משתמשים שזהותם אינה מאומתת ואשר מנסים לגשת ליישום, היישר אל טופס הכניסה.

מימוש של טופס כניסה למטרת אבטחה במסגרת הטופס היא פעולה שכיחה למדי, ולכן מיקרוסופט מימשה בעצמה מספר פקדי Login כדי לפשט את התהליך. כעת תשתמש באחד מהפקדים הללו.

בנה את טופס הכניסה

1. בתפריט Website בחר Add New Item כדי לפתוח את תיבת הדו-שיח Add New Item. ודא שהתבנית Web Form מסומנת ובשדה השם הקלד **LoginForm.aspx**. ודא ששפת התכנות של הטופס היא Visual C#, שתיבת הסימון "Place code in separate file" מסומנת ושתיבת הסימון "Select master page" אינה מסומנת. לחץ Add כדי ליצור את הטופס.

   הטופס החדש ייווצר, וקוד HTML שלו יוצג בחלון תצוגת המקור.
2. עבור לכרטיסייה Design כדי להציג את הטופס LoginForm.aspx בחלון תצוגת העיצוב.

3. בערכת הכלים הרחב את הקטגוריה Login. הוסף לטופס Web פקד **Login**. על הטופס יופיע תפריט Login Tasks. לחץ במקום כלשהו בטופס כדי להסתירו.

הפקד **Login** מורכב ממספר תוויות, שתי תיבות טקסט שמיועדות להזנת שם משתמש וסיסמה, תיבת סימון "remember me" ולחצן כניסה. ניתן להגדיר תצורה של חלק גדול מהמרכיבים באמצעות Properties של הפקד הרלוונטי. ניתן גם להגדיר את סגנון הפקד.

4. גרור את הפקד **Login** למרכז הטופס. לחץ על הסמל Smart Tag שבחלק העליון של הפקד, בסמוך לפינה הימנית. בתפריט Login Tasks שיופיע בחר Auto Format.

כעת תופיע תיבת הדו-שיח Auto Format. ניתן להשתמש בתיבת דו-שיח זו כדי לשנות את המראה והתחושה של הפקד **Login** על ידי בחירת אחת התבניות המוגדרות מראש (ניתן גם להגדיר תבניות מותאמות אישית, באמצעות האפשרות Edit Templates שבתפריט Login Tasks, שמוצגת לאחר לחיצה על Smart Tag שבפקד Login).

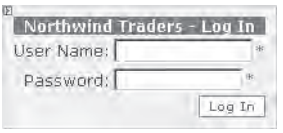
5. בתיבת הדו-שיח Auto Format, בחר בתבנית Classic ולחץ OK. לחץ על הסמל Smart Tag של הפקד **Login** כדי להסתיר את התפריט Login Tasks.

6. בחלון Properties, שנה את מאפייני הפקד **Login** על פי הנתונים המופיעים בטבלה הבאה.

Property	Value
DisplayRememberMe	False
FailureText	Invalid User Name or Password. Please enter a valid User Name and Password.
TitleText	Northwind Traders – Log In
DestinationPageUrl	~/CustomerData.aspx

במאפיין **DestinationPageUrl** עליך לציין את דף האינטרנט שהמשתמש יעבור אליו לאחר שזהותו מאומתת. משמעות הקידומת "~/" היא שהדף נמצא בתיקיית השורש של האתר ולא באחת מתת-התיקיות. כאשר האימות של זהות המשתמש נכשל, תוצג ההודעה שבמאפיין **FailureText**, והמשתמש ייאלץ לחזור על התהליך שנית.

הפקד Login צריך להיראות כך:



כאשר המשתמש לוחץ על הלחצן Log In, יש לאמת את זהותו. אם שם המשתמש והסיסמה תקפים, הוא אמור להמשיך לטופס CustomerData. אחרת, צריכה להופיע הודעת השגיאה המאוחסנת במאפיין **FailureText** של הפקד **Login**. כיצד ניתן לממש את הפעולות הללו?

יש לפחות שתי דרכים לעשות זאת:

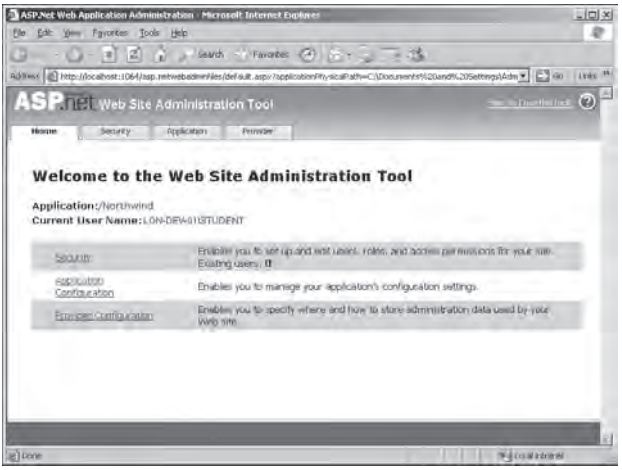
- לכתוב קוד אשר מטפל באירוע Authenticate של הפקד **Login**. אירוע זה מוצף (raised) בכל פעם שהמשתמש לוחץ Login. תוכל לבדוק את הערכים שבמאפיינים **UserName** ו-**Password**, ואם הם תקפים, תוכל לאפשר למשתמש להמשיך לדף שבמאפיין **DestinationPageUrl**. דרך זו למימוש האימות גמישה מאוד, אבל עליך לנהל בעצמך רשימה מאובטחת של שמות משתמשים וסיסמאות.
- ניתן להשתמש בתכונות המובנות של Visual Studio 2005 בשילוב עם ASP.NET Web Site Administration Tool כדי לשלוט בשמות המשתמש והסיסמאות, ולאפשר לפקד **Login** ליישם את תהליכי ברירת המחדל שלו כדי לאמת את זהות המשתמשים בעת לחיצה על הלחצן Log In. ASP.NET Web Site Administration Tool מנהל באופן עצמאי מסדי נתונים של שמות משתמשים וסיסמאות, וכולל אשף שמאפשר להוסיף משתמשים חדשים.

בתרגיל הבא עליך להשתמש באפשרות השנייה.

הגדר את התצורה של אבטחת האתר, והפעל את האבטחה במסגרת הטופס

1. בתפריט WebSite בחר ASP.NET Configuration.

כתוצאה יופעל ASP.NET Development Server לצד בלון המציג את השם URL שלו - הפקודה ASP.NET Configuration תגרום להפעלתו של יישום בשם ASP.NET Web Site Administration Tool אשר משתמש במופע של השרת ASP.NET Development Server, ולכן הוא פועל ללא תלות ביישום הרשת שלך. חלון Internet Explorer ייפתח ויציג את ASP.NET Web Site Administration Tool.



כלי זה כולל מספר דפים שונים המאפשרים להוסיף משתמשים חדשים ולערוך משתמשים קיימים, לשנות הגדרות יישום שברצונך לאחסן בקובץ התצורה ולקבוע את אופן האחסון של נתוני אבטחה, כגון שמות משתמשים וסיסמאות. על פי ברירת המחדל, ASP.NET Web Site Administration Tool מאחסן את נתוני האבטחה בקובץ מסד נתונים של

SQL Server מקומי בשם ASPNETDB.MDF אשר הכלי יוצר בתיקייה App_Data (באמצעות ספק מסד נתונים בשם AspNetSqlProvider) אשר שייכת לאתר האינטרנט. ניתן גם לבחור בספקים אחרים של מסדי נתונים ולאחסן את נתוני האבטחה במיקומים אחרים, אולם נושא זה אינו נדון בספר זה.

.2 עבור לכרטיסייה Security.

כעת יופיע הדף Security. דף זה משמש לניהול המשתמשים, לבחירת מנגנון האימות של האתר, להגדרת תפקידי המשתמשים (תפקידים (roles) משמשים להקצאת סמכויות במסגרת האתר לקבוצה מסוימת של משתמשים), ולבחירת תפקידי גישה אשר משמשים לשליטה בגישה לאתר האינטרנט.

.3 בסעיף Users בחר בקישור "Select authentication type".

כעת יופיע דף חדש שבו תתבקש לבחור כיצד המשתמשים ייכנסו לאתר שלך. בפניך שתי אפשרויות: "From the internet", או "From a local network".

לפי אפשרות "From a local network", האתר ישתמש במנגנוני האימות של Windows. כלומר, כל המשתמשים חייבים להיות חברים רשומים ב- Windows domain שאתר האינטרנט רשאי לגשת אליו. הגישה לאתר Northwind תתבצע דרך האינטרנט, ולכן אפשרות זו אינה רלוונטית למקרה שלנו.

.4 בחר באפשרות "From the internet". אפשרות זו קובעת שהיישום ישתמש באימות במסגרת הטופס (Form-based authentication). כעת תוכל להשתמש בטופס הכניסה (Login) שיצרת בתרגילים הקודמים כדי לבקש מהמשתמש להזין שם וסיסמה. לחץ Done.

חזור לדף Security.

.5 שים לב שבסעיף Users, מספר המשתמשים הנוכחי שיכולים לגשת ליישום הוא אפס. לחץ על הקישור Create User. כתוצאה יופיע הדף Create User.

.6 בדף Create User, הוסף משתמש חדש על פי הערכים שמופיעים בטבלה הבאה:

Prompt	Response
User Name	**John**
Password	**Pa$$w9rd**
Confirm Password	**Pa$$w9rd**
E-mail	**john@northwindtraders.com**
Security Question	**What was the name of your first pet**
Security Answer	**Thomas**

הערה

יש למלא את כל השדות שבמסך זה. הדואר האלקטרוני, שאלת הביטחון ותשובת הביטחון, משמשים את הפקד **PasswordRecovery** כדי לשחזר או לאפס את סיסמת המשתמש. הפקד **PasswordRecovery** נמצא בקטגוריה Login אשר בערכת הכלים, וניתן לצרפו לדף כניסה כדי לעזור למשתמשים אשר שכחו את סיסמתם.

7. ודא שתיבת הסימון Active User אכן מסומנת ולחץ Create User. דף חדש יופיע ובו תוצג ההודעה "Complete. Your account has been successfully created".

8. לחץ Continue. הדף Create User יופיע שוב כדי שתוכל להוסיף משתמשים. לחץ Back כדי לחזור לדף Security. מספר המשתמשים הקיימים עומד כעת על אחד.

הערה

ניתן להשתמש בקישור Manage users שבדף זה כדי לשנות את כתובות הדואר האלקטרוני של המשתמשים ולהוסיף להם תיאור, וכדי להסיר משתמשים קיימים. כדי לאפשר למשתמשים לשנות את סיסמתם, או לשחזרה אם שכחו אותה, הוסף את הפקדים **ChangePassword** ו-**PasswordRecovery** לדף הכניסה של אתר האינטרנט. למידע נוסף בנושא, ראה "Walkthrough: Creating a Web Site with Membership and User Login" ב- Microsoft Visual Studio 2005 Documentation.

9. בסעיף Access Rules לחץ "Create access rules".

כתוצאה יופיע הדף Add New Access Rule. בדף זה עליך לציין מי מהמשתתפים יכולים לגשת לתיקיות מסוימות באתר האינטרנט.

10. בסעיף "Select a directory for this rule", לחץ על התיקייה Northwind כדי לסמן אותה. תחת "Rule applies to", ודא שהשדה user מסומן והקלד **John**. תחת "Permission" לחץ Allow. לחץ OK.

חוק זה מאפשר למשתמש John לגשת לאתר האינטרנט. כעת תחזור למסך Security.

11. בסעיף Access Rules, לחץ שוב על "Create access rules". בדף Add New Access Rule, תחת "Select a directory for this rule", ודא שהתיקייה של Northwind אכן מסומנת. תחת "Rule applies to" לחץ Anonymous user. תחת "Permission" ודא שהאפשרות Deny אכן מסומנת. לחץ OK.

חוק זה מבטיח שמשתמשים שלא נכנסו למערכת דרך "הדלת הראשית" (מסך ההזדהות, login) לא יוכלו לגשת לאתר האינטרנט. המסך Security יופיע שוב.

12. סגור את חלון Internet Explorer אשר מציג את ASP.NET Web Site Administration Tool וחזור לסביבת הפיתוח.

13. לחץ Refresh בסרגל הכלים של Solution Explorer. הקובץ של מסד הנתונים, ASPNETDB.MDF, יופיע בתיקייה App_Data, בשעה שהקובץ Web.config יופיע בתיקיית הפרויקט. לחץ לחיצה כפולה על Web.config כדי להציגו בחלון Code and Text Editor.

קובץ זה נוצר על ידי הכלי ASP.NET Web Site Administration Tool ועליו להיראות כך:

```
<?xml version="1.0" encoding="utf-8"?>
<configuration xmlns="http://schemas.microsoft.com/.NetConfiguration/v2.0">
   <system.web>
      <authorization>
         <allow users="John" />
         <deny users="?" />
      </authorization>
      <authentication mode="Forms" />
   </system.web>
</configuration>
```

האלמנט **<authorization>** מפרט את המשתמשים שיש להם הרשאת גישה, ואחריהם מוצגים המשתמשים שאין להם גישה לאתר האינטרנט (סימן שאלה, **?**, מייצג משתמשים אנונימיים). התכונה **mode** של האלמנט **<authentication>** קובעת שאתר זה משתמש באימות במסגרת הטופס.

14. שנה את האלמנט **<authentication>** והוסף לו תת-אלמנט (child element) **<forms>** באופן הבא. אל תשכח להוסיף גם את האלמנט **</authentication>**:

```
<authentication mode="Forms">
   <forms loginUrl="LoginForm.aspx" timeout="5"
         cookieless="AutoDetect" protection="All" />
</authentication>
```

מטרת האלמנט **<forms>** היא להגדיר את הפרמטרים של האימות במסגרת הטופס. התכונות המפורטות לעיל קובעות, שכאשר משתמש שזהותו לא אומתה מנסה לפתוח את אחד מדפי אתר האינטרנט, הוא ינותב לדף הכניסה, LoginForm.aspx. כאשר משתמש איננו פעיל במשך חמש דקות או יותר, יהיה עליו להיכנס שוב למערכת בפעם הבאה שירצה לגשת לאחד מדפי האתר. בחלק גדול מהאתרים אשר משתמשים באימות במסגרת הטופס, נתוני המשתמש מאוחסנים בקובץ cookie במחשב המשתמש. אולם, מרבית הדפדפנים מאפשרים למשתמש לבטל את השימוש בקבצי cookies (אתרים מסחריים רבים משתמשים בקבצים אלו למטרות זדוניות ולכן לעתים קרובות קבצי cookies נחשבים לסיכונים ביטחוניים). הקוד **cookieless="AutoDetect"** מאפשר לאתר להשתמש בעוגיות כאשר הדפדפן של המשתמש אינו חוסם את האפשרות. אחרת, נתוני המשתמשים מועברים הלוך ושוב בין השרת למחשב הלקוח כחלק מכל בקשה. נתוני המשתמש הם בעצם שם המשתמש והסיסמה. כמובן שאינך מעוניין שמידע זה יהיה זמין לכל, לכן כדאי להשתמש בתכונה **protection** כדי להצפין את הנתונים, כפי שמוצג בדוגמה זו.

15. בתפריט Debug בחר Start Without Debugging.

כעת ייפתח החלון Internet Explorer. דף הפתיחה של היישום הוא CustomerData.aspx, אולם מכיוון שטרם נכנסת למערכת, סביבת ההרצה תנתב אותך לטופס הכניסה LoginForm.

16. הקלד שם משתמש וסיסמה אקראיים, ולחץ Log In.
טופס הכניסה יופיע שוב ויציג את ההודעה "Invalid User Name or Password. Please enter a valid User Name and Password".

17. בשדה User Name הקלד **John**. בשדה Password הקלד **Pa$$w9rd**. לחץ Log In.
כעת יופיע הדף Customer Data עם ההודעה "This form will be implemented later".

18. סגור את Internet Explorer וחזור לסביבת הפיתוח.

תחקור הנתונים

לאחר שלמדת כיצד לשלוט בגישה ליישומים שלך, תוכל להקדיש את תשומת לבך לתחקור ותחזוקה של הנתונים. תצטרך להשתמש בפקדי Web Server Data כדי להתחבר למסד הנתונים, לחקור נתונים ולתחזק אותם. הטכניקה דומה לזו ששימשה אותך לבניית היישום של Windows Forms בפרק 24.

הצגת נתוני הלקוח

בתרגיל הבא עליך לשלוף את כל השורות שבטבלה Customers ממסד הנתונים Northwind שנמצא בשרת SQL Server, ולהציגן בפקד **GridView**. משימתך הראשונה היא ליצור חיבור בין היישום למסד הנתונים Northwind.

הערה
תוכל להשלים תרגיל זה רק לאחר סיום כל התרגילים שבפרק 23.

צור חיבור למסד הנתונים Northwind

1. הצג את הדף CustomerData.aspx בחלון תצוגת העיצוב. מחק את התווית שמציגה את הכיתוב "This form will be implemented later".

2. בערכת הכלים הרחב את הקטגוריה Data. הוסף פקד **SqlDataSource** בטופס.

3. לחץ על הטופס כדי להסתיר את תפריט **SqlDataSource Tasks**. פקד בשם **SqlDataSource1** יתווסף כעת. הפקד SqlDataSource הוא מסוג Web Server אשר נועד למלא את התפקיד שמבצע מקור המידע (data source) ביישומי Windows Forms.

הערה
למרות שהפקד **SqlDataSource** מוצג על הטופס בסביבת העיצוב, הוא לא מוצג כאשר תפעיל אותו בסביבת ההרצה.

4. היעזר בחלון Properties כדי לשנות את המאפיין (ID) של **SqlDataSource1** ל-**CustomerInfoSource**.

5. בחר בפקד **CustomerInfoSource** שעל הטופס. לחץ על הסמל Smart Tag כדי לפתוח את התפריט SqlDataSource Tasks ובחר Configure Data Source.

כתוצאה יופיע האשף Configure Data Source. אשף זה דומה מאוד (אך לא זהה) לאשף שהשתמשת בו בפרק 23. עליך להשתמש באשף זה כדי ליצור חיבור למסד הנתונים וכדי להעביר נתונים מהטבלה Customers.

.6 לחץ על הלחצן New Connection. השתמש בדף Add Connection כדי ליצור חיבור חדש על פי הערכים שבטבלה הבאה. לסיום לחץ OK.

Prompt	Response
Data source	**Microsoft SQL Server (SqlClient)**
Server name	***YourServer*\SQLExpress**
Log on to the server	**Use windows authentication**
Select or enter a database name	**Northwind**

.7 באשף Configure Data Source, לחץ Next.

.8 במסך Save the Connection String to the Application Configuration file, שמור את מחרוזת ההתחברות תחת השם **NorthwindConnectionString** ולחץ Next.

.9 בדף Configure the Select Statement, ודא שהאפשרות "Specify columns from a table or view" מסומנת. ברשימה הנפתחת Name בחר בטבלה Customers. בתיבת הרשימה Columns, סמן "*".

.10 לחץ Advanced. בתיבת הדו-שיח Advanced SQL Generation Options, סמן את האפשרות "Generate INSERT, UPDATE, and DELETE statements." לחץ OK ואחר לחץ Next.

הערה

גם אם לא תסמן את "Generate INSERT, UPDATE, and DELETE statements." תוכל לשנות את הנתונים שנמצאים באובייקט DataSet, ואשר נשלפו על ידי מקור הנתונים, אבל לא תוכל לשלוח את העדכונים בחזרה למסד הנתונים. גם לאחר יצירת מקור הנתונים תוכל להוסיף פקודות לעריכת מסד הנתונים, על ידי שינוי המאפיינים **DeleteQuery**, **InsertQuery** ו-**UpdateQuery** והוספת משפטי SQL מתאימים.

.11 בדף Test Query לחץ Test Query.

הנתונים המאוחסנים בטבלה Customers יופיעו בתיבת הדו-שיח.

.12 לחץ Finish.

.13 לחץ על הסמל Smart Tag כדי להסתיר את התפריט SqlDataSource Tasks.

בתרגיל הבא עליך להוסיף פקד GridView לטופס CustomerData ולכרוך אותו למקור הנתונים CustomerInfoSource.

עצב את הטופס CustomerData

.1 בערכת הכלים, לחץ על הפקד **GridView** וגרור אותו לטופס. לאחר מיקום הפקד לחץ על הטופס כדי להעלים את התפריט GridView Tasks.

פקד **GridView** יתווסף לטופס ויציג נתונים שומרי מקום (placeholder). שנה את גודל הפקד GridView כך שיתפוס את מרבית שטח הטופס.

2. היעזר בחלון Properties כדי לשנות את המאפיין (ID) של הפקד GridView ל-**CustomerGrid**.

3. כאשר הפקד **GridView** עודנו מסומן, לחץ על הסמל SmartTag כדי להציג את התפריט GridView Task ובחר Auto Format.

4. בתיבת הדו-שיח Auto Format, בחר בתבנית Classic, ולחץ OK.

טיפ

אם אף לא אחת מהתבניות המובנות מוצאת חן בעיניך, תוכל לשנות את סגנון המרכיבים השונים של הפקד **GridView** באופן ידני דרך המאפיינים שבחלון Properties. המאפיינים השימושים ביותר הם **BackColor**, **BorderStyle**, **BorderWidth**, **FooterStyle**, **HeaderStyle** ו-**RowStyle**.

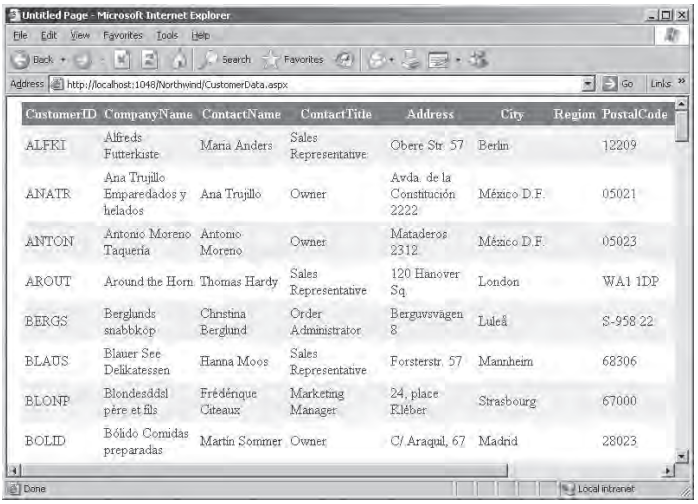
5. בתפריט GridView Tasks, בחר CustomerInfoSource מהרשימה Choose Data Source.
כעת הפקד **GridView** שעל המסך יציג את הכותרות של עמודות הטבלה Customers.

6. לחץ על הסמל Smart Tag כדי להסתיר את התפריט SqlDataSource Tasks.

בדוק את הטופס CustomerData

1. בתפריט Debug בחר Start Without Debugging.
חלון Internet Explorer ייפתח ויציג את הדף Log In.

2. היכנס בתור **John** עם הסיסמה **Pa$$w9rd**.
כעת יופיע הדף CustomerData שמציג את נתוני כל הלקוחות שבמסד הנתונים:



CustomerID	CompanyName	ContactName	ContactTitle	Address	City	Region	PostalCode
ALFKI	Alfreds Futterkiste	Maria Anders	Sales Representative	Obere Str. 57	Berlin		12209
ANATR	Ana Trujillo Emparedados y helados	Ana Trujillo	Owner	Avda. de la Constitución 2222	México D.F.		05021
ANTON	Antonio Moreno Taquería	Antonio Moreno	Owner	Mataderos 2312	México D.F.		05023
AROUT	Around the Horn	Thomas Hardy	Sales Representative	120 Hanover Sq.	London		WA1 1DP
BERGS	Berglands snabbköp	Christina Berglund	Order Administrator	Berguvsvägen 8	Luleå		S-958 22
BLAUS	Blauer See Delikatessen	Hanna Moos	Sales Representative	Forsterstr. 57	Mannheim		68306
BLONP	Blondesddsl père et fils	Frédérique Citeaux	Marketing Manager	24, place Kléber	Strasbourg		67000
BOLID	Bólido Comidas preparadas	Martín Sommer	Owner	C/ Araquil, 67	Madrid		28023

שים לב שנכון לעכשיו, דף זה מאפשר קריאה בלבד. אינך יכול לערוך אף אחד מהנתונים המוצגים. בשלבים הבאים נשפר את הטופס כדי שהמשתמש גם יוכל לבצע שינויים.

3. סגור את Internet Explorer וחזור לסביבת הפיתוח.

אבטחת אתרי אינטרנט ושרתי SQL

כאשר אתה מפעיל יישומים שמשתמשים באבטחה במסגרת הטופס בשרת Development Server של ASP.NET, היישום פועל עם החשבון המשמש אותך להפעלת Visual Studio 2005. בהנחה שאתה משתמש באותו חשבון גם כדי להפעיל את מסד הנתונים Northwind, לא צריכות להיות בעיות כלשהן בעת הגישה של יישום הרשת אל מסד הנתונים.

עם זאת, לאחר שתעביר את אתר האינטרנט לשרת IIS, המצב ישתנה. שרת IIS מפעיל יישומים אשר משתמשים באבטחה במסגרת הטופס באמצעות החשבון של ASPNET. על פי ברירת המחדל (ועקב סיבות אבטחה), לחשבון זה אין גישה ליותר מדי מקומות. הבעיה העיקרית מבחינתנו היא, שדרך חשבון זה לא ניתן להתחבר אל SQL Server Express ולחקור את מסד הנתונים Northwind. לכן, עליך לאפשר לחשבון ASPNET גישה אל SQL Server Express ולהוסיף אותו כמשתמש למסד הנתונים Northwind. לפרטים נוספים בנושא זה עיין בהסבר על הפקודות **sp_grantlogin** ו-**sp_grantdbaccess** שבתיקייה MSDN Library for Visual Studio 2005.

תצוגת הנתונים בדפים

יש תועלת רבה באחזור הנתונים של כל הלקוחות, אבל נניח שיש כמות גדולה מאוד של נתונים בטבלה Customer. אין זה סביר שהלקוח ירצה לעיין בבת אחת בכמות גדולה מאוד של נתונים, כי הוא אינו יכול וגם לא רוצה לטפל בנתונים רבים שאינם דרושים לו. כמו כן, הפקת דף ארוך במיוחד אשר מציג את כל הנתונים, תהיה בזבוז של זמן ושל משאבי רוחב-פס (bandwidth). כדי לפתור את הבעיה יש להציג את המידע על פני מספר דפים, ולאפשר למשתמש לסקור את הדפים השונים בנפרד. זה מה שעליך לעשות בתרגיל הבא.

שימוש בדפדוף בפקד GridView

1. ודא שהקובץ CustomerData.aspx מופיע בחלון תצוגת העיצוב. בחר בפקד **CustomerGrid** ובחלון Properties שנה את המאפיין **AllowPaging** ל-True. כותרת תחתונה שמכילה צמד מספרי דפים תיווסף לפקד **GridView**. כותרת תחתונה זו נקראת **בורר דף** (**pager**). ניתן להגדיר את בורר הדף במספר תצורות. הסגנון הנוכחי הוא סגנון ברירת המחדל, שבו הבורר מורכב ממספרי דפים שהמשתמש יכול ללחוץ עליהם.

2. בחלון Properties, הצב 8 במאפיין **PageSize**.

פעולה זו תגרום לפקד **GridView** להציג שמונה שורות בכל דף.

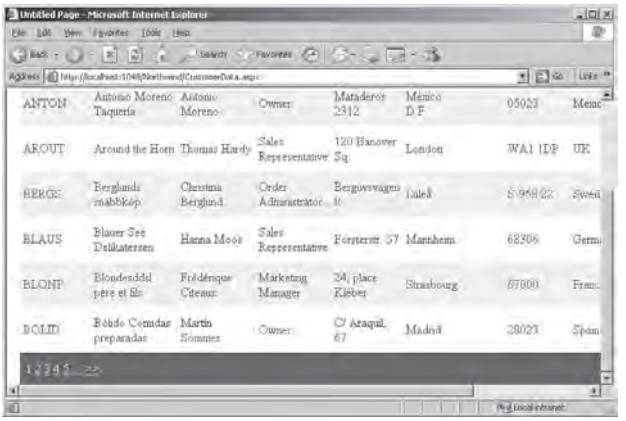
3. הרחב את המאפיין המורכב **PagerStyle**, שבעזרתו ניתן לבחור את התצורה של בורר הדף. בתת-מאפיין **HorizontalAlign** בחר **Left**.

המספרים שבבורר הדף יעברו לצידו השמאלי של הפקד **GridView**.

4. הרחב את המאפיין המורכב **PagerSetting**. באמצעות הערכים שבמאפיין זה ניתן להגדיר את תצורת קישורי הניווט של הדף (page navigation links). יש שתי דרכים להגדיר את קישורי הניווט: בתור מספרי דפים, או בתור שני חצים שמייצגים את הדף הבא (next) והדף הקודם (previous). את המאפיין **Mode** שנה ל-**NumericFirstLast** להצגת מספרי הדפים בשילוב חצים המייצגים את הדף הראשון (first) ואת הדף האחרון (last) כדי לאפשר למשתמש לעבור במהירות לתחילת הנתונים או לסופם. בתת-המאפיין **PageButtonCount** הצב 5; פעולה זו תגרום שקישורים בדף יופיעו בקבוצות של חמשה (תבין את משמעות הדבר כאשר תפעיל את יישום הרשת).

   אם ברצונך להשתמש בחצים של הדף הבא/הקודם, עליך לשנות את טקסט ברירת המחדל (">" ו-"<") על ידי שינוי הערכים שבמאפיינים **NextPageText** ו-**PreviousPageText**. תוכל גם לשנות את הכיתוב של הקישורים לדף הראשון והאחרון על ידי עריכת המאפיינים **FirstPageText** ו-**LastPageText**. שים לב שמאפיינים אלה מקבלים ערכים בפורמט HTML, אחרת הם לא יוצגו כהלכה (לדוגמה, כדי להציג את הסמל ">" עליך לכתוב ">"). ניתן גם להציב שם של קובץ תמונה במאפיינים **NextPageImageUrl**, **PreviousPageImageUrl**, **FirstPageImageUrl** ו-**LastPageImageUrl**. קישור הניווט יופיע כלחצן הנראה כמו תמונה, אם הדפדפן אכן תומך בתמונה מסוג זה.

5. הפעל את יישום הרשת. לאחר הכניסה למערכת, תופענה שמונה שורות הנתונים הראשונות, ומקבץ של קישורי דף יופיע בתחתית הטופס CustomerData. הדפים 1, 2, 3, 4 ו-5 יופיעו לצד הקישור "<<" המשמש כדי לקפוץ לדף האחרון. לחיצה על הקישור "..." תגרום להצגת חמשת מספרי הדפים הבאים לצד הקישור ">>" המשמש לחזרה אל הדף הראשון. דרך הקישור "..." ניתן לחזור לחמשת הדפים הקודמים.



6. לחץ על הקישורים שבתחתית הטבלה כדי לעבור מדף לדף.
7. לסיום סגור את חלון Internet Explorer וחזור לסביבת הפיתוח.

שיפור הגישה לנתונים

בפרק זה הפעלת את הפקד **SqlDataSource** כדי להתחבר למסד הנתונים ולהעביר את הנתונים. מאחורי הקלעים, הפקד **SqlDataSource** יוצר אובייקט DataSet. כאשר אתה כורך פקד DataGrid למקור הנתונים, סביבת הפיתוח מחוללת קוד אשר ממלא את האובייקט DataSet ומציג את הנתונים בפקד.

אובייקטים מסוג DataSet הם בעלי עוצמה רבה. בפרקים הקודמים ראית כיצד ניתן להשתמש בהם כמחסן נתונים בזיכרון המחשב וכיצד ניתן לעדכן באמצעותם את מסד הנתונים. אולם, לאובייקטים מסוג DataSet יש גם חסרונות. DataSet אשר מכיל מספר רב של שורות הוא אובייקט גדול מאוד ולכן עשוי לגזול משאבים רבים. אם אתה משתמש בפקד **GridView** רק כדי להציג נתונים ולא כדי לערוך אותם, השימוש באובייקט DataSet עשוי להיות מיותר. הפקד **SqlDataSource** כולל מאפיין בשם **DataSourceMode** המאפשר לקבוע את סוג אובייקט הנתונים שיוצר **DataSet** (ברירת המחדל) או **DataReader**. אם תבחר באפשרות **DataReader**, מקור הנתונים ייצור אובייקט **DataReader** של ADO.NET אשר ישמש לקריאת הנתונים מהמסד. אובייקט **DataReader** מממש מנגנון יעיל להעברת מידע בשטף (stream) לקריאה בלבד. למידע נוסף בנושא קרא את המסגרת "Firehose Cursors" שבפרק 23. עם זאת, אחד החסרונות של **DataReader** הוא בכך שאינו תומך בדפדוף (paging).

הטמנת נתונים (caching) במקור נתונים

DataSet מכיל עותק של הנתונים שהוא מעביר. ככל שהאובייקט DataSet קיים יותר זמן, הנתונים המאוחסנים בו יהפכו למיושנים יותר. אם כך, כיצד ניתן לוודא שהנתונים המוצגים בפני המשתמש בטופס הם הנכונים, מבלי לחזור ולמלא שוב ושוב את האובייקט DataSet? אם תבחן מקרוב את הפקד **SqlDataSource**, תראה שהוא כולל שלושה מאפיינים שאמורים לסייע בפתרון הבעיה:

- **EnableCaching.** בכל פעם שתציג טופס Web שמכיל את הפקד **SqlDataSource**, יופעל המשפט SELECT אשר מוגדר במסגרת הפקד, כדי למלא את האובייקט DataSet בנתונים. על פי ברירת המחדל, אם אתה משתמש בדפדוף, והשורות מוצגות על פני מספר דפים, המשפט SELECT יופעל בכל פעם שהמשתמש עובר דף. המשפט SELECT יופעל שוב גם אם המשתמש ירענן (refresh) את תצוגת הטופס בדפדפן. כך ניתן לדעת בוודאות שהנתונים המופיעים בעת הצגת הטופס אמנם נכונים ומעודכנים. אבל, אם אף אחד מהנתונים לא התעדכן בין ההפעלות של המשפט SELECT, הרי שבזבזת את משאבי מסד הנתונים כדי לבצע מספר פניות שלא חידשו דבר. אם תשנה את ערך המאפיין **EnableCaching** ל-**True**, האובייקט DataSet ישמש כמטמון והמשפט SELECT יופעל שוב בהתאם להגדרת שני המאפיינים האחרים: **CacheDuration** ו-**CacheExpirationPolicy**.
- **CacheDuration.** מאפיין זה קובע את תדירות ההפעלה החוזרת של המשפט SELECT ורענון המטמון. ערך ברירת המחדל שלו הוא **Infinite** ומשמעותו היא, שתוקף המטמון לעולם אינו פג ולכן גם אין צורך לרענן אותו. הצבה של ערך נומרי במאפיין זה תגרום להגדרת פרק הזמן (בשניות) בין הפעלות של המשפט SELECT.

- **CacheExpirationPolicy**. מאפיין זה משמש בשילוב עם המאפיין **CacheDuration** כדי לקבוע את תדירות עדכון המטמון. אם הערך של מאפיין זה הוא Absolute (ברירת המחדל), יתבצע רענון של המטמון בסופו של כל פרק זמן אשר הוגדר במאפיין **CacheDuration**. אם מצב המאפיין הוא Sliding, רענון המטמון יבוצע רק לאחר חוסר פעילות במסגרת היישום במשך מספר השניות שבמאפיין **CacheDuration**.

חשוב

ניתן לשנות את ערך המאפיין **EnableCaching** ל-True רק אם המאפיין **DataSourceMode** של הפקד **SqlDataSource** נמצא במצב **DataSet**. אם המאפיין **DataSourceMode** מקובע על **DataReader**, היישום יזרוק חריג בעת הצגת הדף.

טיפ

המטמון שימושי גם לצורכי הגנה בפני נפילות זמניות של מסד הנתונים. אם המטמון אינו פעיל, הפקד **SqlDataSource** חייב להיות מחובר למסד הנתונים כל עוד הדף מוצג. אם מסד הנתונים אינו זמין, חיבור זה ייכשל והפקד SqlDataSource יזרוק חריג. אבל כאשר המטמון פעיל, וטרם פג התוקף של הנתונים המאוחסנים בו, תוכל לעבור לדף אחר ולהציג את הנתונים המוטמנים. אם המאפיינים **CacheDuration** ו-**CacheExpirationPolicy** מקובעים לערכים המתאימים, ייתכן שמסד הנתונים לא יהיה זמין לפרק זמן קצר והמשתמש אפילו לא יבחין בכך.

בתרגיל הבא תראה מהן ההשפעות של שינוי הגדרות המטמון של הפקד **SqlDataSource**.

למד כיצד פועל המטמון בשילוב אובייקט SqlDataSource

1. ודא שהקובץ CustomerData.aspx מוצג בחלון תצוגת העיצוב. סמן את הפקד **CustomerInfoSource**. בחלון Properties שנה את ערך המאפיין **EnableCaching** ל-**True**. ודא שהמאפיין **CacheDuration** נקבע ל-**Infinite** ושהמאפיין **CacheExpiration** נקבע ל-**Absolute**.

   שילוב זה גורם לאובייקט DataSet אשר חולל הפקד **SqlDataSource** להתנהג כמטמון שתוקפו לעולם אינו פג. המטמון מתמלא כאשר טופס Web מוצג בפעם הראשונה, אולם לעולם לא יתעדכן כאשר המשתמש עובר בין הדפים או מרענן את הטופס.

2. הפעל את יישום הרשת. היכנס למערכת והצג את הדף הראשון של נתוני הלקוח.

   שים לב שהערך המופיע בעמודה City של השורה הראשונה (ALFKI) הוא Berlin.

3. מבלי לסגור את היישום, פתח את חלון Command Prompt. בחלון הפקודות הקלד את הפקודה הבאה:

```
sqlcmd -S YourServer\SQLExpress -E
```

במקום **YourServer** הקלד את שם המחשב שלך. פקודה זו מפעילה את כלי שורת הפקודה של SQL Server אשר מאפשר להתחבר למאגר הנתונים ולהפעיל משפטי SQL.

הכלי sqlcmd יגרום להופעת המנחה 1> על המסך.

4. לאחר המנחה <1, הקלד את המשפטים הבאים (המנחה ישתנה לאחר כל הקשת Enter):

```
USE Northwind
GO
UPDATE Customers SET City = 'Bonn' WHERE CustomerID = 'ALFKI'
GO
```

כעת תופיע ההודעה **(1 row affected)**. הפקודה תגרום לשינוי ערך העמודה City של הלקוח הראשון מ-Berlin ל-Bonn.

5. מבלי לסגור את חלון הפקודות, חזור לחלון שפועל בו יישום הרשת. עבור לדף 2 וחזור לדף 1. שים לב ששם העיר שבשורה הראשונה לא השתנה - הוא עדיין Berlin. סגור את יישום הרשת וחזור לסביבת הפיתוח.

6. סמן שוב את הפקד **CustomerInfoSource**. בחלון Properties שנה את המאפיין **Duration** ל-10 (ודא גם שהמאפיין **EnableCaching** עדיין **True**).

כעת הפקד CustomerInfoSource מחולל DataSet אשר תוקפו פג כל 10 שניות, ואז האובייקט מעודכן. תוכל לראות את העדכונים אם תציג את הדף פעם נוספת לאחר עדכון האובייקט.

7. הפעל שוב את יישום הרשת. היכנס למערכת והצג את הדף הראשון של נתוני הלקוח. שים לב שהערך בעמודה City של השורה הראשונה (ALFKI) הוא Bonn (זהו השינוי שביצעת קודם לכן).

8. חזור לכלי sqlcmd הפועל בחלון הפקודות. לאחר הסימון <1, הקלד את המשפטים הבאים:

```
UPDATE Customers SET City = 'Munich' WHERE CustomerID = 'ALFKI'
GO
```

ההודעה **(1 row affected)** תוצג שוב. הפקודה תשנה את ערך העמודה City של הלקוח הראשון מ-Bonn ל-Munich.

9. המתן לפחות 10 שניות וחזור לחלון Internet Explorer שבו פועל יישום הרשת. עבור לדף 2 וחזור לדף 1. שים לב שהעיר בשורה הראשונה השתנתה ל-Munich, מכיוון שהאובייקט DataSet עבר רענון. סגור את היישום.

10. סגור את חלון הפקודות.

עריכת הנתונים

ראית כיצד להשתמש בפקד **GridView** כדי להעביר מידע ולעיין בו. התרגילים הבאים עוסקים בעיקר במחיקה ובעריכה של נתונים באמצעות הפקד **GridView**.

מחיקת שורות באמצעות הפקד GridView

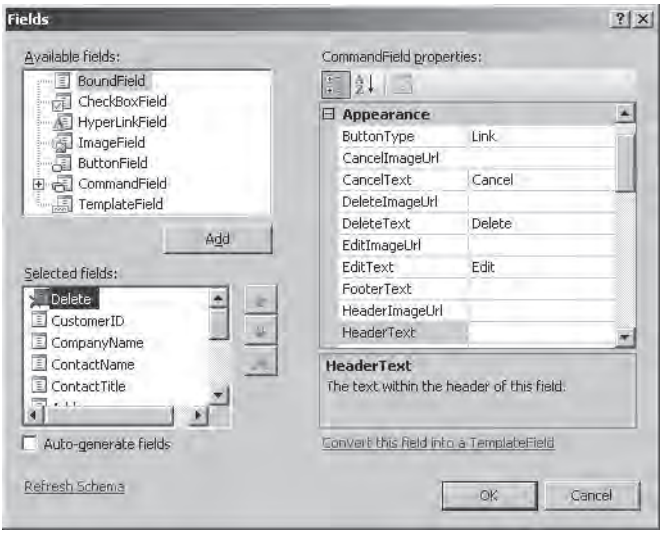
הפקד **GridView** מאפשר להוסיף לו לחצנים המשמשים להפעלת פקודות. באפשרותך להוסיף לחצנים ופקודות שיצרת בעצמך, אבל סביבת הפיתוח כוללת מספר לחצנים מוגדרים מראש המשמשים למחיקה ועריכה של נתונים. בתרגיל הבא עליך להוסיף לחצן Delete לפקד GridControl.

צור לחצן Delete

1. ודא שהקובץ CustomerData.aspx מופיע בחלון תצוגת העיצוב. לחץ על הסמל SmartTag כדי להציג את התפריט GridView Tasks.
2. בתפריט GridView Tasks סמן את תיבת הסימון EnableDeleting.

   היפר-קישור Delete יתווסף כעת לתחילת כל אחת מהשורות שבפקד **GridView**.
3. בתפריט GridView Tasks לחץ Edit Columns.

   תיבת הדו-שיח Fields תופיע על המסך. ניתן להשתמש בתיבת הדו-שיח הזו כדי לשנות את ערכי המאפיינים של השדות (עמודות) המופיעים בפקד **GridView**.



4. ברשימה Selected fields בחר בשדה Delete. ברשימה CommandField properties, שנה את המאפיין **ButtonType** ל-**Button**. לחץ OK.

   הקישור Delete שבפקד **GridView** יהפוך ללחצן.

5. הפעל את היישום. היכנס למערכת ועבור לדף 3 של הנתונים. מחק את הלקוח שהקוד שלו הוא FISSA. המחיקה אמורה להצליח. נסה למחוק את הלקוח FAMIA. פעולה זו תיכשל ותגרום להופעת הודעת שגיאה, כי ללקוח זה יש הזמנות שטרם נשלחו. על פי חוקי היחס של מסד הנתונים Northwind Trader אסור למחוק לקוח שיש לו הזמנות שטרם נשלחו.

טיפ

החריג המופיע אינו ידידותי במיוחד למשתמש (למרות שהוא יכול לעזור מאוד למפתח). כאשר טופס Web זורק חריג, רצוי להציג הודעת שגיאה ידידותית יותר על ידי הפניית המשתמש לדף אחר באמצעות התכונה **ErrorPage** בשילוב עם ההנחיה **@Page** בהגדרות המקור של הקובץ:

```
<%@ Page … ErrorPage="ErrorPage.aspx" %>
```

בדף זה תוכל להציג בפני המשתמש הודעות מועילות יותר.

6. סגור את חלון Internet Explorer וחזור לסביבת הפיתוח.

עדכון שורות באמצעות הפקד GridView

ניתן גם להוסיף לחצן Edit לפקד **GridView**, כדי לאפשר למשתמש לשנות את הנתונים שבשורות השונות המופיעות בפקד. כאשר המשתמש לוחץ Edit, השורה הנבחרת הופכת למקבץ של פקדי **TextBox**. המשתמש יכול לשמור את השינויים או לזנוח אותם. פעולות אלו מתאפשרות על ידי הוספת שני הלחצנים Update ו–Cancel. בתרגילים הבאים עליך להוסיף את שלושת הלחצנים הללו לטופס CustomerData.

הוסף את הלחצנים Update, Edit ו–Cancel

1. הצג את הטופס CustomerData.aspx בתצוגת העיצוב. לחץ על הסמל Smart Tag של הפקד **GridView** כדי לפתוח את התפריט GridView Tasks, ולאחר מכן לחץ Enable Editing.

   לחצן Edit יתווסף לצד כל אחת מהשורות שבפקד **GridView**.

הערה

סגנון הלחצן Edit זהה לזה של הלחצן Delete. אם הלחצן Delete עדיין מופיע כהיפר-קישור, כך גם יופיע הלחצן Edit.

2. הפעל את היישום. היכנס למערכת ולאחר מכן לחץ על הלחצן Edit עבור השורה הראשונה שבטופס CustomerData.

   השורה הראשונה תהפוך לאוסף פקדי **TextBox**, והלחצנים Edit ו–Delete יוחלפו בלחצנים Update ו–Cancel.

הערה

העמודה CustomerID נשארת כמו תווית, כי עמודה זו הינה המפתח העיקרי (primary key) של הטבלה Customers. המשתמש אינו אמור לשנות את ערכי המפתח העיקרי במסד הנתונים מכיוון שהדבר עלול לפגוע באחידות ובעקביות של הטבלאות.

3. שנה את הנתונים בעמודות ContactName ו-ContactTitle, ולחץ Update.

מסד הנתונים יעודכן, השורה תהפוך שוב למקבץ של תוויות, הלחצנים Edit ו-Delete יופיעו שוב, והנתונים החדשים יוצגו בשורה.

4. סגור את חלון Internet Explorer וחזור לסביבת הפיתוח Visual Studio 2005.

☯ **אם ברצונך להמשיך לפרק הבא**

השאר את Visual Studio 2005 פעילה, ועבור לפרק 28.

☯ **אם ברצונך לכבות כעת את Visual Studio 2005**

פתח את תפריט Menu ולחץ Exit. אם מופיעה תיבת דו-שיח Save, לחץ Yes.

פרק 27 – טבלה מסכמת

המשימה	צריך
ליצור טופס כניסה (Login) לאתר.	צור טופס Web חדש. הוסף לטופס פקד **Login** שמטרתו אימות זהותו של המשתמש.
לקבוע את ההגדרות של אבטחת אתר אינטרנט של ASP.NET.	השתמש בכלי ASP.NET Web Site Administration Tool כדי להוסיף משתמשים ולערוך את נתוניהם, להגדיר תפקידים (roles), וליצור תפקידי גישה (access roles).

לכלול מנגנון אבטחה במסגרת הטופס (Form-based security).	ערוך את הקובץ Web.config. קבע את התכונה **<authentication mode>** ל-**Forms**, ציין את הכתובת URL של טופס הכניסה, והעבר את הפרמטרים הנחוצים לתהליך האימות. לדוגמה:
	`<authentication mode="Forms">`
	`  <forms loginUrl="LoginForm.aspx"`
	`  timeout="5"`
	`  cookieless="AutoDetect"`
	`  protection="All" />`
	`</authentication>`

ליצור טופס Web המשמש להצגת נתונים ממסד נתונים.	הוסף לטופס פקד מקור נתונים והגדר אותו באופן שיתחבר למסד הנתונים הרצוי. הוסף פקד **GridView** לטופס, ובמאפיין **DataSourceID** בחר את הפקד שמשמש כמקור הנתונים.
העבר והצג נתונים במקטעים קצרים בטופס Web.	שנה את המאפיין **AllowPaging** של הפקד **GridView** לערך **True**. במאפיין **PagerSize** הצב את מספר השורות שברצונך להציג בכל דף. שנה את המאפיינים **PagerSetting** ו-**PagerStyle** כדי להתאים את סגנון הפקד לזה של הטופס.
להשתמש במטמון בשילוב עם פקדי מקור נתונים.	שנה את המאפיין **EnableCaching** של פקד מקור הנתונים לערך True. קבע את הזמן בשניות שלוקח לתוקף המטמון לפוג באמצעות המאפיין **CacheDuration**. קבע האם פרק זמן זה הוא אבסולוטי או יחסי באמצעות המאפיין **CacheExpirationPolicy**.
לערוך ולמחוק שורות במסד נתונים בעזרת פקד **GridView**.	ודא שהאפשרות "Generate Insert, Update, and Delete statements" מסומנת בעת יצירת מקור הנתונים באמצעות האשף Configure Data Source. לחילופין, השתמש באשף Configure Data Source כדי להוסיף משפטי INSERT, UPDATE ו-DELETE למקור נתונים קיים. בתפריט GridView Tasks (נפתח על ידי לחיצה על הסמל Smart Tag), סמן את Enable Update ו- Enable Deleting. בתפריט Edit Columns בחר את סגנון הצגת הקישורים Update ו-Delete בפקד **GridView**.

פרק 28

יצירה והפעלה של שירותי רשת (Web services)

כאשר תסיים פרק זה, תוכל:

- ליצור **שירות רשת** (Web Service) החושף שיטות פעולה פשוטות.
- לבדוק שירות רשת בעזרת דפדפן Internet Explorer.
- לעצב מחלקות שניתן להעביר בתור פרמטרים לשיטות, ואשר שיטות יכולות להחזיר.
- ליצור הפניה לשירות רשת ביישום הלקוח.
- להפעיל שיטה משירות רשת.

בפרקים הקודמים למדת ליצור טפסי Web ולבנות יישומי רשת אינטראקטיביים בעזרת ASP.NET. למרות שגישה זו טובה למצבים שבהם הלקוח הוא דפדפן, לעתים קרובות תיתקל גם במצבים אחרים. כפי שנאמר בפרקים קודמים, האינטרנט היא למעשה רשת מאוד גדולה. ניתן לבנות **מערכות בזורות** (distributed systems) מרכיבים שנמצאים במספר מקומות ברשת – מסדי נתונים, שירותי אבטחה, רכיבים פיננסיים וכד'. בפרק זה תלמד לעצב, לבנות ולבדוק רכיבים שהגישה אליהם מתאפשרת באמצעות האינטרנט, ולאחר מכן לאחד אותם ליישום מרכזי אחד. תלמד גם לבנות יישום שמפעיל ומנצל שיטות שנחשפות על ידי שירות רשת.

מהו שירות רשת?

שירות רשת (Web service) הינו רכיב עסקי המספק שירותים שימושיים ללקוחות, או לצרכנים (consumers). ניתן לראות בשירות רכיב רגיל בעל נגישות גלובלית. שירותי רשת משתמשים בפרוטוקול HTTP (פרוטוקול סטנדרטי, מקובל וברור) כדי לשלוח נתונים, וגם בפורמט נתונים נייד המתבסס על XML. יש תקן שימוש עבור HTTP ו–XML, ולכן ניתן להשתמש בהם גם בסביבות תכנות אחרות מלבד NET Framework. משמעות הדבר היא שניתן לבנות שירות רשת באמצעות Visual Studio 2005, ולבנות יישומי לקוח (צרכנים) שפועלים בסביבת עבודה שונה לחלוטין, כגון Java. התאמה זו היא דו–כיוונית: ניתן לבנות שירות רשת באמצעות Java, ולכתוב את יישום הצרכן בשפת #C.

ניתן להשתמש במספר שפות בשילוב עם Visual Studio 2005 כדי לבנות שירותי רשת. נכון לעכשיו, השפות הנתמכות הן Microsoft Visual C++, Microsoft Visual C#, Microsoft Visual J# ו- Microsoft Visual Basic .NET, ויש להניח שרשימה זו תגדל בעתיד. מבחינת הצרכן אין חשיבות לשפת התכנות שבאמצעותה הוקם שירות הרשת, וגם לא מעניינו כיצד השירות מבצע את המוטל עליו. מבחינתו, שירות הרשת הוא ממשק שחושף מספר שיטות שמטרתן ברורה. הצרכן צריך לקרוא לשיטות אלו על ידי שימוש בפרוטוקולים סטנדרטיים של אינטרנט, ולהשתמש בפורמט XML כדי להעביר את הפרמטרים ולקבל את המענה.

אחד הכוחות המניעים שמאחורי NET Framework. ומהדורות עתידיות של Windows, הוא תפישת "programmable Web" (רשת שניתנת לתכנות). הרעיון הוא שבעתיד המערכות יורכבו מנתונים ושירותים שמספקים מספר שירותי רשת. שירותים אלה יספקו את המרכיבים הבסיסיים עבור המערכות, ספקי האינטרנט יספקו את האמצעי לגשת אליהם, והמפתחים - הם ישלבו את הכל יחדיו בצורה הגיונית. שירותי הרשת הם הבסיס של יישומים רבים מסוג **עסק-לעסק** (B2B, Business to Business) ומסוג **עסק-לצרכן** (B2C, Business to Client).

התפקיד של SOAP

SOAP (Simple Object Access Protocol) הוא פרוטוקול שמשמש את הצרכן כדי לשלוח לשירות הרשת בקשות ולקבל ממנו תגובות או מענה.

SOAP הוא פרוטוקול פשוט שבנוי על בסיס HTTP, שמאפשר להעביר את ההודעות שלו גם באמצעות פרוטוקולים אחרים, אבל נכון לעכשיו, רק השילוב בין HTTP ו-SOAP הוא היחיד שהוגדר כתקני. הפרוטוקול מגדיר כללים לפורמט XML לציון שמות השיטות שהצרכן רוצה להפעיל בשירות הרשת, ולהגדרה ולתיאור של הפרמטרים והערכים המוחזרים. כאשר לקוח קורא לשירות, עליו לציין את השיטות והפרמטרים על פי כללי התחביר של XML.

SOAP הוא התקן של התעשייה. מטרתו לשפר את יכולת התקשורת בין מערכות שונות. יתרונותיו של SOAP הם הפשטות והעובדה שהוא מתבסס על טכנולוגיות שמהוות גם הן תקן תעשייתי: HTTP ו-XML.

המפרט של SOAP מגדיר מספר דברים, שהחשובים מביניהם:

- המבנה של הודעת SOAP.
- כיצד לקודד את הנתונים.
- כיצד לשלוח הודעות (קריאה לשיטות).
- כיצד לעבד את התשובות.

ניקח לדוגמה שירות רשת בשם ProductService.asmx (ב- .NET Framework), שמות URL עבור שירותי רשת מקבלים את הסיומת asmx) אשר חושף שיטות המשמשות לעבודה עם נתוני הטבלה Products שבמסד הנתונים Northwind (מייד נבנה שירות רשת כזה). אחת השיטות היא **HowMuchWillItCost**, אשר מאפשרת ללקוח להעביר שם של מוצר וכמות. השיטה חוקרת את מחיר המוצר במסד הנתונים כדי לחשב את העלות הכוללת של ההזמנה. בקשת SOAP שנשלחת על ידי הלקוח עשויה להיראות כך:

```
POST /NorthwindServices/Service.asmx HTTP/1.1
Host: localhost
Content-Type: application/soap+xml; charset=utf-8
Content-Length: 579

<?xml version="1.0" encoding="utf-8"?>
<soap12:Envelope xmlns:xsi="http://www.w3.org/2001/XMLSchema-
instance" xmlns:xsd="http://www.w3.org/2001/XMLSchema" xmlns:
soap12="http://www.w3.org/2003/05/soap-envelope">
   <soap12:Body>
      <HowMuchWillItCost
         xmlns="http://www.contentmaster.com/NorthwindServices">
         <productName>Chai</productName>
         <howMany>39</howMany>
      </HowMuchWillItCost>
   </soap12:Body>
</soap12:Envelope>
```

בקשה זו מורכבת משני חלקים: כותרת שכוללת את כל הקוד עד התגית **<soap12:body>**, וגוף ההודעה שנמצא במסגרת התגית **<soap12:body>**. ניתן לראות את אופן קידוד הפרמטרים בגוף ההודעה. בדוגמה זו שם המוצר הוא Chai והכמות היא 39.

שרת הרשת יקבל את הבקשה, יזהה את שירות הרשת ואת השיטה שעליו להפעיל, יפעיל את השיטה, יקבל את התוצאות, וישלח אותן חזרה ללקוח כתוצאת SOAP, כך:

```
HTTP/1.1 200 OK
Content-Type: application/soap+xml; charset=utf-8
Content-Length: 546

<?xml version="1.0" encoding="utf-8"?>
<soap12:Envelope xmlns:xsi="http://www.w3.org/2001/XMLSchema-
instance" xmlns:xsd="http://www.w3.org/2001/XMLSchema" xmlns:
soap12="http://www.w3.org/2003/05/soap-envelope">
   <soap12:Body>
      <HowMuchWillItCostResponse
         xmlns="http://www.contentmaster.com/NorthwindServices">
         <HowMuchWillItCostResult>529</HowMuchWillItCostResult>
      </HowMuchWillItCostResponse>
   </soap12:Body>
</soap12:Envelope>
```

הלקוח יכול כעת לשלוף את התוצאה מגוף ההודעה ולעבד אותה.

מהי שפת תיאור שירותי רשת?

גוף הודעת SOAP מנוסח בתבנית XML. השרת מצפה מהלקוח להשתמש במספר תגיות כדי לקודד את הפרמטרים שהוא מעביר לשיטה. כיצד הלקוח יודע באיזו תבנית עליו להשתמש? התשובה היא, שהשירות אמור להעביר תיאור של עצמו כאשר הוא מתבקש לעשות זאת. לקוח יכול לשלוח בקשה לשירות הרשת המכילה את המחרוזת עם **WSDL** (**שפת תיאור שירותי רשת**, Web Services Description Language):

```
http://localhost/NorthwindServices/Service.asmx?wsdl
```

תגובת שירות הרשת תהיה תיאור שדומה לזה:

```
<?xml version="1.0" encoding="utf-8"?>
<wsdl:definitions xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
    ...
    targetNamespace="http://www.contentmaster.com/NorthwindServices"
    xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/">
<wsdl:types>
   <s:schema elementFormDefault="qualified"
    targetNamespace="http://www.contentmaster.com/orthwindServices">
     <s:element name="HowMuchWillItCost">
      <s:complexType>
       <s:sequence>
        <s:element minOccurs="0" maxOccurs="1"
                name="productName" type="s:string" />
        <s:element minOccurs="1" maxOccurs="1"
                name="howMany" type="s:int" />
       </s:sequence>
      </s:complexType>
</s:element>
<s:element name="HowMuchWillItCostResponse">
   <s:complexType>
      <s:sequence>
        <s:element minOccurs="1" maxOccurs="1"
                name="HowMuchWillItCostResult" type="s:decimal" />
        </s:sequence>
       </s:complexType>
     </s:element>
    </s:schema>
   </wsdl:types>
...
</wsdl:definitions>
```

הודעה זו נקראת **WSD** (Web Service Description) (הסרנו חלק גדול מההודעה כדי לחסוך במקום), והתבנית המשמשת לכתיבתה נקראת **WSDL** (Web Service Description Language). תיאור זה כולל מידע שמאפשר ללקוח לבנות בקשות SOAP בתבנית ששירות הרשת יכול להבין. התיאור לכאורה מסובך, אבל למרבה המזל, סביבת הפיתוח Visual Studio 2005 מספקת כלים שבאמצעותם ניתן לנתח בקלות את תיאור WSDL של השירות, ולאחר מכן להשתמש בו כדי ליצור אובייקט מורשה (proxy) שמאפשר ללקוח להמיר קריאות של שיטות לבקשות SOAP. נעשה זאת בהמשך הפרק, לאחר שתלמד כיצד לבנות **שירות רשת**, או **שירות אינטרנט** (Web Service).

שיפור שירותי הרשת

זמן קצר לאחר ששירותי הרשת (**שירותי האינטרנט**) הפכו לטכנולוגיה מקובלת לחיבור בין שירותים פזורים, היה ברור שיש סוגיות שגם SOAP וגם HTTP אינם יכולים לפתור בעצמם. סוגיות אלו כוללות:

- **אבטחה**. כיצד ניתן להבטיח שאף אחד אינו לוכד ועורך את הודעות SOAP המועברות בין שירותי הרשת לבין הצרכן ברחבי האינטרנט? כיצד ניתן לדעת בוודאות שהודעת SOAP אכן נשלחה על ידי הצרכן או שירות הרשת שטוענים ששלחו אותה, ולא על ידי אתר שהוקם למטרות איסוף מידע במרמה? כיצד ניתן להגביל את הגישה לשירותי הרשת? נושאים אלה חשובים מבחינת אמינות ההודעה, הסודיות והאימות, ועליהם להיות בראש מעייניו של המפתחים וכל העוסקים בבניית יישומים מבוזרים שפועלים באמצעות האינטרנט.

  בשנות ה-90 המוקדמות, מספר חברות המספקות כלים להקמת מערכות מבוזרות הקימו ארגון שבשלב מאוחר יותר קיבל את השם Organization for the Advancement of Structured Information Standards, או **OASIS**. עם הבנת החסרונות שבתשתית של שירותי הרשת המוקדמים, חברי ארגון OASIS דנו בבעיות אלו וניסחו את המפרט WS-Security, אשר מציג את הדרכים לאבטח הודעות שנשלחות על ידי שירותי רשת. החברות אשר מנויות ל-WS-Security צריכות לממש את דרישות המפרט, בעיקר על ידי יישום מנגנון של **תעודות הסמכה** (certificates) ומנגנוני הצפנה.

- **מדיניות (Policy)**. למרות שהדרישות של WS-Security תורמות לשיפור האבטחה, המפתחים עדיין צריכים לכתוב קוד שיממש את המפרט. לעתים קרובות, שירותי רשת אשר נוצרו על ידי מפתחים שונים מממשים מנגנוני הגנה ברמות שונות. לדוגמה, שירות רשת עשוי להשתמש בהצפנה פשוטה למדי שניתן לפצח בקלות יחסית, בעוד שצרכן אשר שולח מידע אישי חסוי לשירות רשת ידרוש ככל הנראה רמת הצפנה גבוהה יותר. זו דוגמה אחת למדיניות. יש דוגמאות נוספות, כגון איכות השירות ואמינות שירות הרשת. שירות רשת יכול לממש רמות שונות של אבטחה, איכות, שירות ואמינות, ולחייב את הלקוח בהתאם. יישום הלקוח ושירות הרשת יכולים לקבוע יחדיו מהי רמת האבטחה הרצויה על פי דרישות ועלויות. אולם, כדי לקבל החלטה כזו, הלקוח ושירות הרשת חייבים להיות בעלי הבנה משותפת של המדיניות הקיימת. המפרט WS-Policy כולל מודל יעדים כלליים ותחביר תואם, שבאמצעותו ניתן לתאר ולהעביר את המדיניות של שירות הרשת.

- **ניתוב והפניה לכתובות (routing and addressing)**. חשוב ששרת הרשת יוכל לנתב מחדש בקשת שירות רשת לאחד מהשירותים. לדוגמה, מערכות רבות מיישמות חלוקת-עומס (load-balancing). הבקשות שנשלחות לשרת מנותבות מחדש על ידי השרת אל מחשבים אחרים כדי לחלק את העומס ביניהם.

יש מספר אלגוריתמים שיכולים לשמש את השרת לחלוקת העומס. הדבר החשוב הוא שהלקוח אשר שלח את הבקשה אינו מודע לניתובה מחדש. ניתוב בקשות שירות רשת דרוש במיוחד כאשר מנהל השרת צריך לכבות את אחד המחשבים לצרכי אחזקה. בקשות שבדרך כלל היו מופנות למחשב הכבוי, מנותבות לאחד המחשבים האחרים. המפרט WS-Addressing מתאר מסגרת לניתוב בקשות משירותי רשת.

מהן ההשלכות של כל הסוגיות האלו בעת פיתוח שירותי רשת באמצעות Visual Studio 2005? ובכן, מיקרוסופט מימשה בעצמה את המפרטים WS-Security, WS-Policy ו–WS-Addressing כחלק מחבילת WSE (Web Service Enhancements). ניתן להוריד את החבילה בחינם מאתר Microsoft שכתובתו היא **http://msdn.microsoft.com/webservices/building/wse/default.aspx**. כאשר אתה מתקין את החבילה, היא משתלבת באופן אוטומטי בסביבת Visual Studio ומוסיפה רכיבים ותבניות פרויקט.

עבור Visual Studio 2005 עליך להוריד את WSE 3.0.

הערה

אין צורך להתקין את WSE 3.0 כדי לבצע את התרגילים שבפרק זה.

בניית שירות הרשת ProductService

בפרק זה עליך ליצור את השירות ProductService, אשר יחשוף שתי Web methods. השיטה הראשונה מאפשרת למשתמש לחשב את העלות הכוללת של קניית כמות מסוימת של מוצר כלשהו ממסד הנתונים Northwind, והשיטה השנייה מקבלת את שם המוצר ומחזירה את פרטיו.

יצירת שירות הרשת ProductService

בתרגיל הראשון עליך ליצור את שירות הרשת ProductService ולכתוב את שיטת הרשת **HowMuchWillItCost**. לאחר מכן עליך לבדוק את השיטה כדי לוודא שהיא פועלת כהלכה.

צור את שירות הרשת

1. בסביבת הפיתוח של Visual Studio 2005, צור אתר אינטרנט חדש בעזרת התבנית ASP.NET Web Service. ודא שהשדה Location נקבע לערך File System ושהשדה Language נקבע לערך Visual C#. צור את אתר האינטרנט בתיקייה \My Documents\Microsoft Press\Visual CSharp Step by Step\Chapter 28\Northwind Traders.

חשוב

שים לב שלא לבחור בטעות בתבנית ASP.NET Web Site במקום בתבנית ASP.NET Web Service.

סביבת הפיתוח תחולל אתר אינטרנט שמכיל תיקיות בשם App_Code ו-App_Data וקובץ בשם Service.asmx. קובץ זה מכיל את הגדרות שירות הרשת. הקוד של השירות מוגדר במסגרת המחלקה **Service**, מאוחסן בקובץ Service.cs ומוצג בחלון Code and Text Editor.

2. ב- Solution Explorer, לחץ על הפרויקט NorthwindServices. בחלון Properties שנה את ערך המאפיין **Use dynamic port** ל-**False**, ובמאפיין **Port number** הצב **4500**.

על פי ברירת המחדל, השרת Development Web server אשר כלול בסביבת הפיתוח, בוחר יציאה (port) באקראי כדי להפחית את הסבירות להתנגשות עם יציאות אחרות המשמשות שירותי רשת אחרים שפועלים במחשב. זהו אלמנט שימושי למקרים שבהם אתה בונה ובודק אתרי אינטרנט של ASP.NET בשרת פיתוח לפני שאתה מעתיק אותם אל שרת ייצור (production server) כגון IIS. אולם, בעת בניית שירותי רשת עדיף להשתמש במספר יציאה קבוע, מכיוון שהיישום של הלקוח צריך להתחבר לשירות.

3. בקובץ Service.cs אשר בחלון Code and Text Editor, עיין במחלקה **Service**. מחלקה זו היא צאצא של המחלקה **System.Web.Services.WebService**. הבט בחלק התחתון של המחלקה. התבנית של סביבת הפיתוח כוללת שיטת שירות בשם **HelloWorld** שמטרתה להדגים את המבנה של שיטה זו. השיטה מחזירה את המחרוזת "Hello World". שים לב שיש לסמן את כל השיטות שהלקוח יכול לקרוא להן בתכונה **[WebMethod]**. בתרגיל זה אין לנו צורך בשיטה, ולכן הפוך אותה להערה.

4. לפני המחלקה Service ניתן להבחין בשתי תכונות נוספות: **[WebService]** ו-**[WebServiceBinding]**. התכונה **[WebServiceBinding]** מזהה את רמת מפרט יכולת העבודה ההדדית של שירותי הרשת (Web services interoperability specification) שעל פיה צריך שירות הרשת לפעול. בפרק זה ניתן להתעלם מתכונה זו ולכן אל תשנה את ערך ברירת המחדל שלה. התכונה **[WebService]** מפרטת את שמות מתחמי השמות (namespaces) המשמשים לזיהוי שירות הרשת. שנה את ערכי התכונה הזו באופן הבא:

```
[WebService(Namespace="http://www.contentmaster.com/
NorthwindServices")]
[WebServiceBinding(ConformsTo = WsiProfiles.BasicProfile1_1)]
public class Service : System.Web.Services.WebService
{
   …
}
```

5. ב- Solution Explorer, לחץ לחיצה ימנית על הפרויקט NorthwindServices ובתפריט הקיצור שייפתח בחר Add New Item. בתיבת הדו-שיח Add New Item, לחץ על התבנית Web Configuration File. ודא ששם הקובץ הוא Web.config ולחץ Add. פעולה זו תגרום להוספת קובץ תצורה שמכיל ערכי ברירת מחדל לפרויקט. הקובץ Web.config ייפתח בחלון Code and Text Editor.

6. שנה את האלמנט **<connectionString/>** שבקובץ Web.config. הוסף את תת-האלמנט **<add>** שלהלן, כדי להגדיר את מחרוזת ההתחברות עבור מסד הנתונים Northwind. במקום **YourServer** כתוב את שם המחשב שלך.

אל תשכח להוסיף בסוף את האלמנט <connectionString/>:

```
<connectionStrings>
   <add name="NorthwindConnectionString"
      connectionString="Data Source=YourServer\SQLExpress; Initial
Catalog=Northwind; Integrated Security=True"
        providerName="System.Data.SqlClient"/>
</connectionStrings>
```

7. הצג את הקובץ Service.cs בחלון Code and Text Editor. הוסף את השיטה הבאה למחלקה **Service**, אחרי השיטה Hello method:

```
[WebMethod]
public decimal HowMuchWillItCost(string productName, int howMany)
{
}
```

שיטה זו אמורה לקבל מהלקוח שם של מוצר המופיע בטבלה Products שנמצאת במסד הנתונים Northwind ואת מספר הפריטים הרצוי. השיטה תשלוף את הנתונים הנחוצים ממסד הנתונים, כדי לחשב את העלות הכוללת של ההזמנה, ותעביר את התוצאה בתור הערך המוחזר של השיטה.

8. הוסף את משפטי **using** הבאים בתחילת הקובץ:

```
using System.Configuration;
using System.Data.SqlClient;
```

מרחב השמות **System.Configuration** מכיל מחלקות המשמשות לקריאת הגדרות התצוגה מהקובץ Web.config. מרחב השמות **System.Data.SqlClient** מכיל מחלקות Microsoft ADO.NET אשר מאפשרות גישה לשרת Microsoft SQL Server.

9. הוסף לשיטת הרשת **HowMuchWillItCost** את המשפטים הבאים:

```
SqlConnection sqlConn = null;

try
{
   ConnectionStringSettings cs =
      ConfigurationManager.ConnectionStrings["NorthwindConnectionString"];
   string connString = cs.ConnectionString;
   sqlConn = new SqlConnection(connString);
   sqlCommand sqlCmd = new SqlCommand();
   sqlCmd.CommandText = "SELECT UnitPrice FROM Products " +
      "WHERE ProductName = '" + productName + "'";
   sqlCmd.Connection = sqlConn;
   sqlConn.Open();
   decimal price = (decimal)sqlCmd.ExecuteScalar();
   return price * howMany;
}
```

```
catch(Exception e)
{
   // Handle the exception
}
finally
{
   if (sqlConn != null)
      sqlConn.Close();
}
```

מטרת הקוד לשלוף את מחרוזת ההתחברות **NorthwindConnectionString** מהקובץ Web.config באמצעות המאפיין **ConnectionString** של המחלקה **ConfigurationManager**. מאפיין זה מחזיר את הרשומה (entry) המקבילה מהקובץ Web.config. רשומה זו מכילה מידע, כגון שם מחרוזת ההתחברות, הספק ומחרוזת ההתחברות עצמה. ניתן לשלוף את מחרוזת ההתחברות מהרשומה באמצעות המאפיין **ConnectionString**.

כעת הקוד יתחבר למאגר הנתונים Northwind ויפעיל את המשפט **SELECT** SQL כדי לשלוף מהטבלה Products את העמודה UnitPrice של המוצר הנבחר. השיטה **ExecuteScalar** היא הדרך הטובה ביותר להפעלת משפטי **SELECT** המחזירים ערך בודד. העמודה UnitPrice תאוחסן במשתנה **price**, אשר לאחר מכן יוכפל בפרמטר **howMany** שהועבר לשיטה כדי לחשב את העלות הכוללת.

הערה

אם שכחת כיצד להשתמש במודל של ADO.NET כדי לגשת למסדי נתונים, עיין שוב בסעיף "יישום ADO.NET באמצעות תכנות" שבפרק 23.

השירות משתמש בבלוק **try/catch** כדי לטפל בשגיאות בסביבת ההרצה, למרות שהשירות אינו מבצע בדיקה של הפרמטרים המועברים לשיטה (לדוגמה, הלקוח עשוי להעביר ערך שלילי לפרמטר **howMany**). תוכל להוסיף בעצמך את הקוד לבדיקת תקינות הפרמטרים.

חשוב מאוד לסגור את החיבור למסד הנתונים גם במקרה של חריג, וזו למעשה מטרת הבלוק **finally**.

אם נזרק חריג, עליך לאחסן את פרטיו. ברוב המקרים לא כדאי להחזיר למשתמש את פרטי החריג אל המשתמש, כי יש סיכוי סביר שהוא לא יבין את משמעותם, וגם יש סיבות של אבטחה. יש חשש שמידע רב מדי על אופן פעולת שירות הרשת ייחשף בפני המשתמש (פרטים כגון השם והמיקום של מסד הנתונים הם בדיוק הפרטים שהאקרים זקוקים להם כדי לפרוץ למערכת). אולם, עבור מנהל אתר הרשת, פרטי החריג הם בעלי חשיבות רבה. המקום האידיאלי לאחסון הפרטים של חריג הנזרק בשירות רשת הוא יומן האירועים (Event Log) של Windows.

10. הוסף את המשפט **using** הבא לראש הטופס:

```
using System.Diagnostics;
```

זהו מרחב השמות המכיל את המחלקות המשמשות לשליטה ביומן האירועים.

11. הוסף את השיטה הבאה למחלקה **Service**. שים לב שזו שיטה פרטית (private) מקומית אשר אינה מסומנת בתכונה **[WebMethod]**:

```
private void handleWebException(Exception e)
{
   EventLog log = new EventLog("Application");
   log.Source = "NorthwindServices";
   log.WriteEntry(e.Message, EventLogEntryType.Error);
}
```

משפטים אלה מוסיפים רשומה ליומן האירועים של Windows Application וכוללים בה את פרטי החריג שהועבר לשיטה כפרמטר; ובנוסף הם מסמנים את החריג כשגיאה (ביומן האירועים ניתן לאחסן גם אזהרות והודעות מידע אחרות). עם זאת, לא כל אחד יכול להוסיף רישומים ביומן האירועים, וכדי לעשות זאת יש לקבל אישור מתאים ממנהל המערכת (Administrator). בתרגיל הבא תלמד כיצד להעניק את ההרשאות הדרושות.

12. בבלוק **catch** של שיטת הרשת **HowMuchWillItCost**, הסר את ההערה והוסף במקומה את המשפטים הבאים. המשפט הראשון קורא לשיטה **handleWebException**, ומעביר לה את החריג שנזרק בתור פרמטר. המשפט **throw** זורק חריג ריק אשר גורם להצגת דף האינטרנט "HTTP 500 - Internal server error" בדפדפן של המשתמש.

```
catch(Exception e)
{
   handleWebException(e);
   throw new Exception();
}
```

13. בתפריט Build, בחר Build Web Site כדי להדר את שירות הרשת.

המבנה של שירות רשת

שירותי רשת ב- NET Framework. דומים ליישומי ASP.NET מהבחינה ששניהם מורכבים משני חלקים: הקובץ asmx אשר מגדיר את שירות הרשת וקובץ של קוד C# עם הסיומת cs. קובץ הקוד cs מכיל למעשה את קוד C# של השיטות שאתה מצרף לשירות הרשת, ועל פי ברירת המחדל הוא מוצג בחלון Code and Text Editor. הקובץ מאוחסן בתיקייה App_Code של אתר האינטרנט, וכדי לראות אותו עליך להרחיב את התיקייה App_Code ב- Solution Explorer.

כדי לצפות בתוכן הקובץ asmx עליך ללחוץ עליו לחיצה כפולה ב- Solution Explorer. קובץ asmx של שירות הרשת NorthwindServices נראה כך:

```
<%@ WebService Language="C#" CodeBehind="~/App_Code/Service.
cs" Class="Service" %>
```

הסבירות שתצטרך לבצע שינויים בקובץ זה היא נמוכה ביותר.

מרחבי שמות של שירותי רשת

לכל שירות רשת צריך להיות מרחב שמות ייחודי נפרד, כדי שיישומי לקוח יוכלו להבדיל בינו לבין שירותי רשת אחרים. על פי ברירת המחדל, שירותי רשת הנוצרים באמצעות Visual Studio 2005 משתמשים בכתובת http://tempuri.org. אין בזה כל קושי כאשר מדובר בשירותי רשת שנמצאים בפיתוח, אבל כאשר הופכים את השירות לציבורי, צריך ליצור עבורו מרחב שמות. בדרך כלל נהוג ליצור את שם מרחב השמות באמצעות כתובת URL של החברה, בשילוב עם מזהה נוסף. מרחב השמות אינו חייב להיות URL קיים, כי כל מטרתו לתת לשירות הרשת זהות ייחודית.

מתן הרשאת גישה ליומן האירועים של Windows

חשוב

עליך לדעת את סיסמת המשתמש של מנהל המערכת במחשב שלך, כדי לבצע את התרגיל הבא.

1. בסביבת Windows, פתח את תפריט Start, All Programs, Accessories ולחץ לחיצה ימנית על Command Prompt. בתפריט הקיצור בחר Run As. בתיבת הדו-שיח Run As סמן את תיבת הסימון The following user. בשדה User name הקלד **Administrator**, הקלד את הסיסמה ולחץ OK.

   כעת יופיע חלון Command Prompt, אשר פועל בתור Administrator, ואתה יכול לעשות בו כל מה שמנהל המערכת יכול - אבל, שים לב והיזהר!

2. בחלון הפקודות הקלד **regedit**.

   כעת יופעל הכלי Registry Editor, אשר מאפשר להגדיר את תצורת המחשב. **עליך להיות זהיר ביותר, מכיוון שאתה יכול לגרום לנזק שכדי לתקנו צריכים לפעמים גם להתקין מחדש את מערכת ההפעלה.**

3. בתצוגת העץ שבחלונית השמאלית, הרחב את התיקייה My Computer\HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Services\Eventlog.

4. לחץ לחיצה ימנית על Application, ובתפריט הקיצור פתח את New ולחץ Key. מפתח חדש בשם New Key #1 יופיע בחלונית השמאלית. שנה את שמו ל-**NorthwindServices**.

טיפ

אם הקלדת שם שגוי תוכל לתקנו. לחץ לחיצה ימנית על המפתח, ובתפריט הקיצור בחר Rename. אל תשנה את שמות המפתחות האחרים.

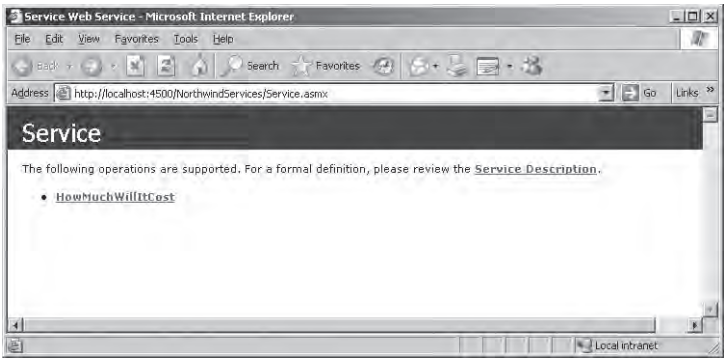
5. סגור את חלון Registry Editor, ולאחר מכן סגור את חלון הפקודות (על ידי הקשת הפקודה Exit).

בדוק את שיטת הרשת

1. חזור לסביבת הפיתוח. ב- Solution Explorer, לחץ לחיצה ימנית על Service.asmx ובתפריט הקיצור בחר View in Browser.

   השרת Development Web Server של ASP.NET יופעל ויציג הודעה שאומרת ששירות הרשת זמין כעת בכתובת http://localhost:4500/NorthwindServices.

   חלון Internet Explorer ייפתח, ויעבור לכתובת URL זו: http://localhost:4500/NorthwindServices/Service.asmx. דף הבדיקה Service יופיע בחלון.



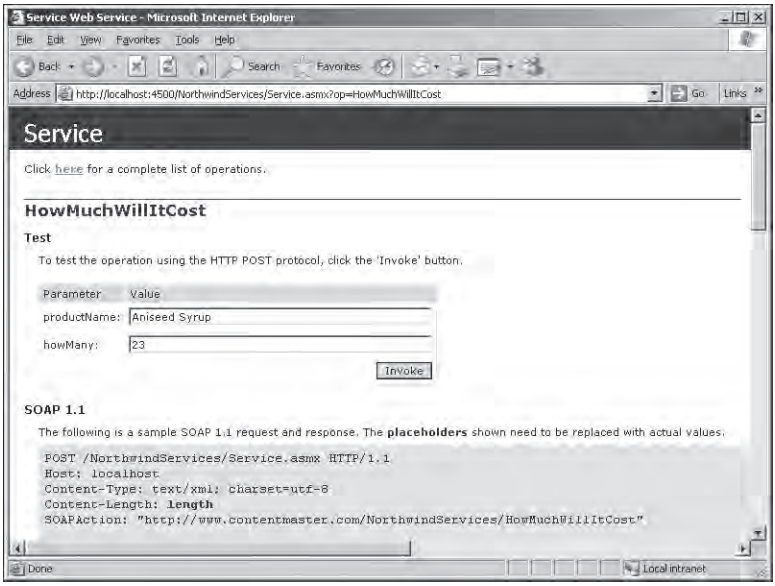
   דף הבדיקה מאפשר לצפות בתיאור WSDL על ידי לחיצה על ההיפר-קישור Service Description, או לבדוק את אחת משיטות הרשת (בדוגמה זו יש שיטה אחת בלבד: **HowMuchWillItCost**).

2. לחץ על ההיפר-קישור Service Description.

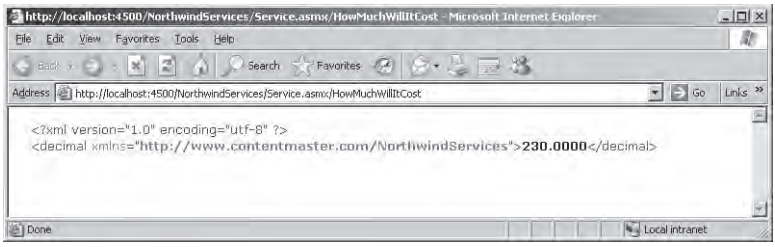
   הכתובת URL תשתנה כעת לכתובת זו: http://localhost:4500/NorthwindServices/Service.asmx?WSDL. בחלון Internet Explorer יופיע התיאור WSDL של שירות הרשת שבנית.

3. לחץ על הלחצן Back שבסרגל הכלים של Internet Explorer כדי לחזור לדף הבדיקה. לחץ על ההיפר-קישור **HowMuchWillItCost**.

   Internet Explorer יחולל כעת דף נוסף המאפשר להעביר פרמטרים אל השיטה **HowMuchWillItCost** כדי לבדוק אותה. הדף כולל גם דוגמאות של בקשת SOAP ומענה SOAP.



4. בתיבת הטקסט **productName** הקלד **Anissed Syrup** ובתיבת הטקסט **howMany** הקלד 23. לחץ Invoke. שיטת הרשת תופעל וחלון Internet Explorer נוסף ייפתח ובו יוצג המענה בפורמט SOAP.



5. סגור את שני חלונות Internet Explorer וחזור לסביבת הפיתוח.

טיפול בנתונים מורכבים

SOAP מאפשר להעביר מבני נתונים מורכבים בין הלקוח לבין השירות בתור פרמטרים של קלט, פרמטרים של פלט או בתור ערכים מוחזרים. כדי לעשות זאת, צריך להמיר את מבני הנתונים לפורמט אשר ניתן לשלוח דרך הרשת ולסדרו מחדש בקצה המקבל. תהליך זה מכונה **serialization** (סיריאליזציה). לדוגמה, בתרגיל הבא עליך לבנות מחלקה שמכילה נתונים על אחד המוצרים שבמסד הנתונים Northwind. היא כוללת מספר מאפיינים וביניהם **ProductID**, **ProductName**, **SupplierID** וגם **CategoryID**. לאחר מכן עליך ליצור שיטה אשר מחזירה מופע של מחלקה זו.

מטרת תהליך הסיריאליזציה של SOAP היא להמיר את המחלקה לקוד XML, לשלוח גרסה זו דרך הרשת באמצעות SOAP, ולבנות מחדש את המחלקה על פי קוד XML בקצה המקבל. המבנה שלהלן ממחיש את אופן ההמרה של המחלקה לפני העברתה:

```
<?xml version="1.0" encoding="utf-8" ?>
<Product xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xmlns:xsd="http://www.w3.org/2001/XMLSchema"
   xmlns="http://www.contentmaster.com/NorthwindServices">
   <ProductID>1</ProductID>
   <ProductName>Chai</ProductName>
   <SupplierID>1</SupplierID>
   <CategoryID>1</CategoryID>
   <QuantityPerUnit>10 boxes x 20 bags</QuantityPerUnit>
   <UnitPrice>18.0000</UnitPrice>
   <UnitsInStock>39</UnitsInStock>
   <UnitsOnOrder>0</UnitsOnOrder>
   <ReorderLevel>10</ReorderLevel>
   <Discontinued>false</Discontinued>
</Product>
```

תהליך הסיריאליזציה הינו אוטומטי ונסתר מעיני המשתמש כל עוד תקיים מספר כללים פשוטים בעת הגדרת המחלקה. ניתן לבצע סיריאליזציה רק עם מחלקות אשר מכילות שדות ומאפיינים ציבוריים בלבד. כאשר האובייקט מכיל חברים פרטיים ללא אחראי גישה (accessors) **set** ו-**get** מתאימים, לא יהיה ניתן להעבירו בצורה נכונה, מכיוון שהנתונים הפרטיים לא יאותחלו בצד המקבל. שים לב שכל המאפיינים חייבים לכלול אחראי גישה **set** ו-**get**. הסיבה לכך היא שכחלק מתהליך הסיריאליזציה לפורמט XML יש לכתוב את הנתונים גם באובייקט, לאחר שהם מועברים. כמו כן, המחלקה חייבת לכלול בנאי ברירת מחדל (שאינו מקבל פרמטרים).

נהוג לעצב מחלקות עבור SOAP ככלים להעברת נתונים. תוכל להגדיר מחלקות פונקציונליות נוספת המשמשות כ-facades ומספקות את הלוגיקה (הקוד) העסקית עבור מבני הנתונים. המשתמשים והיישומים יוכלו לגשת לאותם נתונים באמצעות business facades האלו.

ניתן גם להגדיר את תצורת מנגנון ה-serialization באמצעות מחלקות SOAP השונות שבמרחב השמות **System.Xml.Serialization**, או להגדיר מנגנוני סיריאליזציה של XML על ידי מימוש הממשק **ISerializable** שבמרחב השמות **System.Runtime.Serialization**.

הגדרת המחלקה Product

1. ב- Solution Explorer, לחץ לחיצה ימנית על התיקייה App_Code ובתפריט הקיצור בחר Add New Item. בתיבת הדו-שיח Add New Item, בחר בתבנית Class, ובחר בשם **Product.cs** עבור המחלקה החדשה. לחץ Add כדי ליצור את המחלקה.

2. ב- Solution Explorer, הרחב את התיקייה App_Code אם אינה מורחבת עדיין. לחץ לחיצה כפולה על הקובץ Product.cs כדי להציגו בחלון Code and Text Editor.

3. הוסף את המשתנים הפרטיים (private) הבאים למחלקה **Product**, לפני הבנאי. יש משתנה אחד עבור כל אחת מהעמודות שבטבלה Products שבמסד הנתונים.

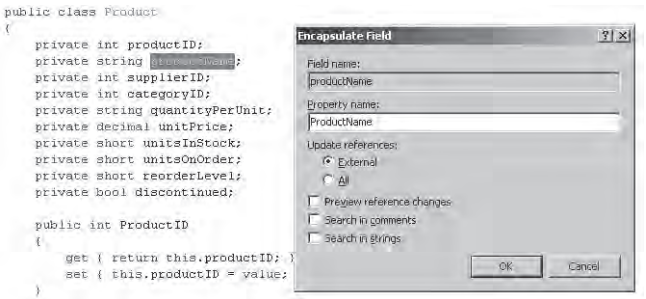
```
private int productID;
private string productName;
private int supplierID;
private int categoryID;
private string quantityPerUnit;
private decimal unitPrice;
private short unitsInStock;
private short unitsOnOrder;
private short reorderLevel;
private bool discontinued;
```

4. צור מאפיין קריאה/כתיבה בשם **ProductID**, שישמש כשם גישה למשתנה **productID**:

```
public int ProductID
{
   get { return this.productID; }
   set { this.productID = value; }
}
```

5. הוסף מאפייני קריאה/כתיבה עבור כל אחד מהמשתנים הנותרים. יש לפחות שתי דרכים להוספת המאפיינים: כתיבה ידנית של כל אחד מאחראי הגישה (accessors) **get** ו-**set**; חילול אחראי הגישה על ידי Visual Studio 2005.

כדי לחולל מאפיין עבור המשתנה **productName** על ידי Visual Studio 2005, לחץ לחיצה כפולה על המשתנה **productName** שבקוד כדי לסמנו. בתפריט Refactor בחר Encapsulate Field. בתיבת הדו-שיח Encapsulate Field ודא שבשדה Property name מופיע ProductName, הסר את הסימון מתיבת הסימון "Preview reference changes" ולחץ OK, כמוצג במסך הבא:



המאפיין הבא יתווסף למחלקה class:

```
public int ProductName
{
   get { return productName; }
   set { productName = value; }
}
```

חזור על התהליך עבור המשתנים האחרים.

שמות של שדות ומאפיינים: אזהרה

למרות שמקובל לתת לשדות פרטים ולמאפיינים שמות זהים, למעט גודל האות הראשונה (A או a, למשל), עליך להיות מודע לחיסרון שבעיקרון זה. הבט בקוד שלהלן:

```
public int CategoryID
{
   get { return this.CategoryID; }
   set { this.CategoryID = value; }
}
```

קוד זה יעבור את תהליך ההידור בהצלחה, אולם התוכנית תיתקע בכל פעם שתתבצע גישה למאפיין **CategoryID**. הסיבה לכך היא שאחראי הגישה get ו-set מפנים למאפיין (C גדולה) במקום לשדה הפרטי (c קטנה), והדבר גורם ללולאה רקורסיבית אינסופית. סוג כזה של באגים קשה מאוד לאיתור.

צור את שיטת הרשת GetProductInfo

1. חזור לקובץ Service.cs בחלון Code and Text Editor. הוסף שיטת רשת נוספת בשם **GetProductInfo** אשר מקבלת כפרמטר את שם המוצר (מחרוזת) ומחזירה אובייקט מסוג **Product**:

```
[WebMethod]
public Product GetProductInfo(string productName)
{
}
```

2. הוסף את המשפטים הבאים לשיטה **GetProductInfo**.

```
Product product = new Product();
SqlConnection sqlConn = null;
try
{
   ConnectionStringSettings cs =
      ConfigurationManager.ConnectionStrings["NorthwindConnectio
       nString"];
   string connString = cs.ConnectionString;
   sqlConn = new SqlConnection(connString);
   SqlCommand sqlCmd = new SqlCommand();
   sqlCmd.CommandText = "SELECT * FROM Products " +
      "WHERE ProductName = '" + productName + "'";
```

```
   sqlCmd.Connection = sqlConn;
   sqlConn.Open();
   SqlDataReader productData = sqlCmd.ExecuteReader();
   if (productData.Read())
   {
      product.ProductID = productData.GetInt32(0);
      product.ProductName = productData.GetString(1);
      product.SupplierID = productData.GetInt32(2);
      product.CategoryID = productData.GetInt32(3);
      product.QuantityPerUnit = productData.GetString(4);
      product.UnitPrice = productData.GetDecimal(5);
      product.UnitsInStock = productData.GetInt16(6);
      product.UnitsOnOrder = productData.GetInt16(7);
      product.ReorderLevel = productData.GetInt16(8);
      product.Discontinued = productData.GetBoolean(9);
   }
   else
   {
      throw new ArgumentException("No such product " +
productName);
   }
   productData.Close();
   return product;
}

catch(ArgumentException e)
{
   handleWebException(e);
   throw e;
}

catch(Exception e)
{
   handleWebException(e);
   throw new Exception();
}

finally
{
   if (sqlConn != null)
       sqlConn.Close();
}
```

משפטים אלו מתחברים למסד הנתונים Northwind על ידי שימוש במודל ADO.NET ושולפים ממנו את הנתונים של המוצר הרלוונטי.

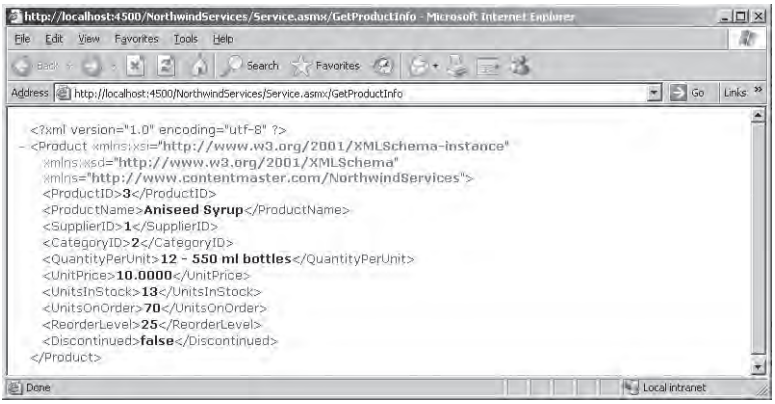
שיטה זו מטפלת במרבית החריגים באופן דומה לשיטה **HowMuchWillItCost**. אולם, אם לא נמצא מוצר הנושא את השם שהועבר לשיטה, חשוב שהלקוח ידע את זה, וניתן להעביר לו את הודעת השגיאה מבלי לפגוע באבטחת היישום. במקרה שכזה תזרוק השיטה חריג **ArgumentException**. כדי שחריג זה לא יסונן על ידי המטפל בחריגים כלליים, המטפל בחריג **ArgumentException** זורק את אותו החריג פעם נוספת ומעביר את נתוני החריג למשתמש.

3. בנה את אתר האינטרנט. ב-Solution Explorer לחץ לחיצה ימנית על Service.asmx ובתפריט הקיצור בחר View in Browser כדי להפעיל את שירות הרשת. לאחר שדף הבדיקה יוצג בחלון Internet Explorer, לחץ על ההיפר-קישור **GetProductInfo**.

דף הבדיקה **GetProductInfo** יופיע ויאפשר לך לבחור את השיטה **GetProductInfo**.

4. בתיבת הטקסט productName הקלד Aniseed Syrup ולחץ Invoke.

שיטת הרשת תופעל, תעביר את נתוני המוצר **Aniseed Syrup** ותחזיר אובייקט מסוג **Product**. אובייקט זה יעבור תהליך סיריאליזציית XML ויוצג בחלון Internet Explorer.



5. סגור את חלון Internet Explorer.

שירותי רשת, לקוחות ואובייקטים מורשים

ראית ששירותי רשת עושים שימוש ב-SOAP כדי לקבל בקשות וכדי לשלוח בחזרה מענים. SOAP ממיר את הנתונים המועברים ל-XML, ומעבירם באמצעות פרוטוקול HTTP המשמש את שרתי האינטרנט ואת הדפדפנים. זוהי הסיבה להצלחתם של שירותי הרשת - HTTP ו-XML ברורים מאוד (לפחות בתיאוריה) ודנים בהם במספר ועדות סטנדרטים. הסטנדרט של SOAP עדיין נמצא בפיתוח, והוא אומץ על ידי מספר חברות אשר מעוניינות לספק שירותים דרך האינטרנט. לקוח אשר "מדבר" בשפת SOAP יכול לתקשר עם שירותי רשת, גם אם הלקוח ושירות הרשת ממומשים באמצעות שפות תכנות שונות לחלוטין, ופועלים במערכות שבדרך כלל אינן יכולות לעבוד זו עם זו. לדוגמה, יישום לקוח שפותח באמצעות Microsoft Visual Basic ופועל במחשב כף-יד, יכול לתקשר עם שירות רשת הפועל במחשב מסוג IBM 390 ומפעיל קוד של UNIX.

אז כיצד יכול הלקוח "לדבר" בשפת SOAP? ישנן שתי דרכים: הדרך הקשה והדרך הקלה.

לדבר SOAP בדרך הקשה

בדרך הקשה על הלקוח לבצע מספר פעולות:

1. לבחור בכתובת URL של שירות הרשת המפעיל את שיטת הרשת.

2. לבצע חקירת WSDL (Web Services Description Language) על ידי שימוש בכתובת URL כדי לקבל תיאור של השיטות הקיימות, הפרמטרים שיש להעביר להן והערכים שהן מחזירות. מסמך זה מתקבל בפורמט XML (דוגמה למסמך שכזה ראית בפרק הקודם).
3. להמיר את כל הקריאות לשיטות הרשת לכתובת URL המתאימה, ולהשתמש בתהליך הסיריאליזציה כדי להמיר כל אחד מהפרמטרים לפורמט המוגדר במסמך WSDL.
4. לשלוח את הבקשה, בשילוב הנתונים שהתקבלו מתהליך הסיריאליזציה, לכתובת URL על ידי שימוש ב-HTTP.
5. להמתין לקבלת מענה משירות הרשת.
6. להשתמש בפורמטים המופיעים במסמך WSDL כדי לבצע תהליך דיסיריאליזציה (de-serialize) על הנתונים שהתקבלו משירות הרשת, כדי לקבל ערכים אותם יכול היישום שלך לעבד.

הפעלת השיטה באופן שכזה מצריכה עבודה רבה, ובמהלך התהליך עשויות להיות שגיאות.

לדבר SOAP בדרך הקלה

החדשות הרעות הן שהדרך הקלה לא יותר מדי שונה מהדרך הקשה. החדשות הטובות הן שניתן לבצע את התהליך באופן אוטומטי, מכיוון שבחלקו הגדול הוא טכני. חברות רבות מפתחות כלים המסוגלים לחולל מחלקות מתווכות (proxy classes) על פי התיאור WSDL. המתווך (proxy) מסתיר את המורכבות של השימוש ב-SOAP וחושף ממשק תכנות פשוט המתבסס על השיטות שבשירות הרשת. יישום הלקוח קורא לשיטה על ידי הפעלת שיטות בעלות שם זהה במתווך. המתווך ממיר את הקריאות לשיטות המקומיות לבקשות SOAP ושולח אותן לשירות הרשת. המתווך ממתין למענה, מבצע תהליך דיסיריאליזציה לנתונים, ומעביר אותם בחזרה ללקוח באותו אופן שבו מוחזר ערך מקריאה רגילה לשיטה.

צריכת שירות הרשת ProductService

בפרק זה יצרת קריאה לשירות רשת אשר חושפת שתי שיטות: השיטה **GetProductInfo** שמטרתה להחזיר את נתוניו של המוצר המועבר כפרמטר, והשיטה **HowMuchWillItCost** שמטרתה לקבוע את העלות הכוללת של קניית n פריטים של המוצר x. בתרגיל הבא, עליך ליישם את שירות הרשת שייצרת וליצור יישום אשר צורך את השיטות הללו. נתחיל עם השיטה **GetProductInfo**.

צור יישום לקוח לשירות רשת

1. פתח עוד מופע של Visual Studio 2005. חשוב לעשות זאת מכיוון ששרת הפיתוח של ASP.NET יפסיק לעבוד אם תסגור את הפרויקט הקודם של שירות הרשת NorthwindServices, ואם זה יקרה לא תוכל לגשת אליו מהשרת (ניתן גם ליצור את יישום הלקוח כפרויקט נוסף בפתרון שבו נמצא שירות הרשת). כאשר תפעיל שירותי רשת בסביבת ייצור (production environment) באמצעות IIS, לא תיתקל בבעיה זו מכיוון שהשרת IIS פועל באופן עצמאי.

2. במופע החדש של Visual Studio 2005, צור פרויקט חדש בעזרת התבנית Windows Application. קרא לפרויקט **ProductInfo** ושמור אותו בתיקייה \My Documents\Microsoft Press\Visual CSharp Step by Step\Chapter 28.

3. שנה את שם הקובץ Form1.cs ל-**ProductForm.cs**.

4. שנה את גודל הטופס ל- 392, 400. את המאפיין **Text** שנה ל-**Product Details**.

5. הוסף לטופס 10 תוויות, וסדר אותם בטור במרווחים שווים בצידו השמאלי של הטופס. מלמעלה למטה, קבע את מאפייני Text של עשרת התוויות על פי הערכים הבאים: Product Name, Product ID, Supplier ID, Category ID, Quantity Per Unit, Unit Price, Units In Stock, Units On Order, Reorder Level ו-Discontinued.

6. הוסף תשע תיבות טקסט לטופס ומקם אותן בסמוך לתשע התוויות הראשונות. אפס את המאפיין **Text** של כל אחת מתיבות הטקסט. מלמעלה למטה, קבע את המאפיין **Name** של כל אחת מתיבות הטקסט על פי הערכים הבאים: **productName**, **productID**, **supplierID**, **categoryID**, **quantityPerUnit**, **unitPrice**, **unitsInStock**, **unitsOnOrder** ו-**reorderLevel**.

7. הוסף לטופס תיבת סימון לצד התוויות **Discontinued** ומתחת לתיבת הטקסט **reorderLevel**. את המאפיין **Name** של הפקד שנה ל-**discontinued** ואפס את המאפיין **Text**.

8. הוסף לטופס לחצן ומקם אותו מימין לתיבת הטקסט **productName**. שנה את שם הלחצן ל-**getProduct**, ושנה את המאפיין **Text** ל-**Get Product**.

בסיום התהליך על הטופס להיראות כך:

הוסף הפניה לשרת הרשת

1. בתפריט Project בחר Add Web Reference.

על המסך תופיע תיבת הדו-שיח Add Web Reference.

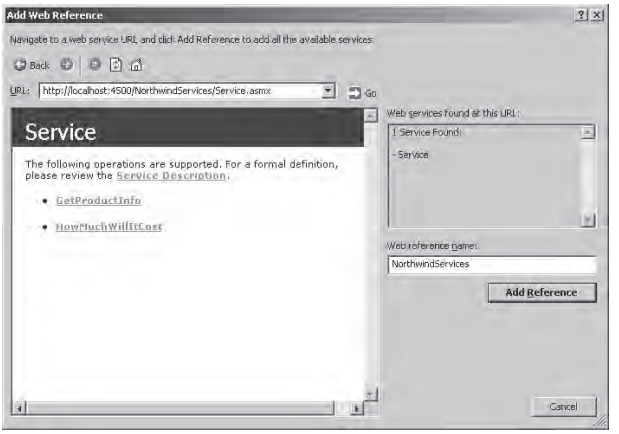
תיבת דו-שיח זו משמשת לחיפוש שירותי רשת ולעיון בתיאורי WSDL.

2. הקלד בתיבת הטקסט Address הנמצאת בראש תיבת הדו-שיח את הכתובת URL של שירות הרשת **http://localhost:4500/NorthwindServices/ :NorthwindTraders Service.asmx**.

טיפ

אם המארח (host) של שירות הרשת הוא השרת ISS, תוכל ללחוץ על ההיפר-קישור "Web services on the local machine" הנמצא בחלונית השמאלית של תיבת הדו-שיח במקום להקליד את הכתובת באופן ידני. במקרה שלנו המארח הוא שרת Development של ASP.NET ולכן הוא לא יופיע כאשר תלחץ על ההיפר-קישור.

כעת יופיע דף הבדיקה של שירות הרשת ויוצגו בו השיטות **GetProductInfo** ו-**HowMuchWillItCost**. שנה את הערך שבתיבת הטקסט Web reference name ל-NorthwindServices, כמוצג במסך הבא:



3. לחץ Add Reference והבט ב- Solution Explorer. נוספה לו תיקייה חדשה בשם Web References המכילה פריט בשם NorthwindServices. לחץ על הפניית הרשת NorthwindServices (Web reference) ועבור לחלון Properties כדי לעיין במאפייניה. שים לב למאפיין Web Reference URL המכיל את כתובת URL של שירות הרשת.

הפעל את שיטת הרשת

1. הצג את הקובץ ProductForm.cs בחלון Code and Text Editor. הוסף לרשימה שבראש הקובץ את משפטי using הבאים:

```
using ProductInfo.NorthwindServices;
```

בעת הוספת הפניית רשת לפרויקט, המתווך (proxy) אותו מחולל שירות הרשת ממוקם במרחב שמות ששמו זהה לזה של ההפניה לשירות הרשת, או במקרה שלנו, NorthwindServices.

2. צור שיטת אירוע עבור האירוע **Click** של הלחצן **getProduct** וקרא לה **getProduct_Click**. בשיטה **getProduct_Click** צור את המשתנה הבא:

```
Service northwindService = new Service();
```

Service היא המחלקה המתווכת המאפשרת גישה לשירות הרשת (שם המתווך יהיה תמיד זהה לשם השירות). המחלקה נמצאת במרחב השמות **NorthwindServices**. כדי להשתמש בשירות עליך ליצור מופע של המחלקה המתווכת, וזה בעצם מה שקוד זה עושה.

3. הוסף קוד שמטרתו להפעיל את שיטת הרשת **GetProductInfo** דרך השיטה **getProduct_Click**. אתה בוודאי מודע לחוסר היציבות של רשתות בכלל ושל האינטרנט בפרט. צור בלוק **try/catch** אחרי המשפט אשר יוצר את המשתנה **northwindService**. זכור שהשירות זורק חריג גם במקרה שהמשתמש מנסה לשלוף פריט שאינו קיים במסד הנתונים.

```
try
{
   // Code goes here in the next steps
}
catch (Exception ex)
{
   MessageBox.Show("Error fetching product details: " +
      ex.Message, "Error", MessageBoxButtons.OK,
      MessageBoxIcon.Error);
}
```

הוסף את המשפט הבא לבלוק **try**:

```
Product prod = northwindService.GetProductInfo(productName.
Text);
```

בזכות האובייקט המתווך (**northwindService**) הקריאה לשיטה **GetProductInfo** נראית כמו קריאה רגילה לשיטה מקומית. המידע המוחזר על ידי השיטה **GetProductInfo** משמש ליצירת מופע של המחלקה **Product**. התיאור WSDL של השירות כולל את המידע הנחוץ להגדרת המבנה של מחלקה זו, וניתן להשתמש בו במסגרת יישום הלקוח באופן שהוצג לעיל.

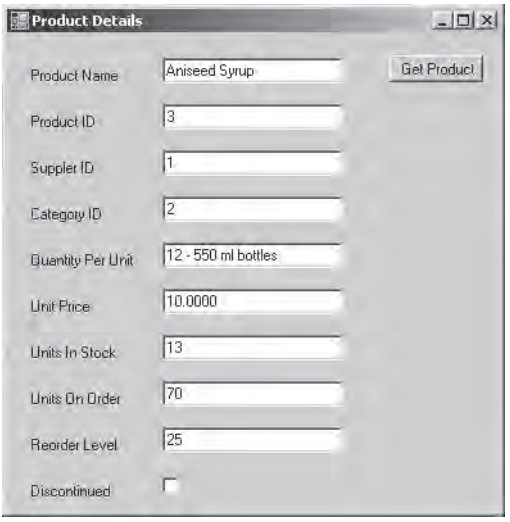
4. הוסף לבלוק try את המשפטים הבאים, שתפקידם לשלוף את נתוני המוצר מהאובייקט Product ולהציגם על גבי הטופס:

```
productID.Text = prod.ProductID.ToString();
supplierID.Text = prod.SupplierID.ToString();
categoryID.Text = prod.CategoryID.ToString();
quantityPerUnit.Text = prod.QuantityPerUnit;
unitPrice.Text = prod.UnitPrice.ToString();
unitsInStock.Text = prod.UnitsInStock.ToString();
unitsOnOrder.Text = prod.UnitsOnOrder.ToString();
reorderLevel.Text = prod.ReorderLevel.ToString();
discontinued.Checked = prod.Discontinued;
```

בדוק את היישום

1. בנה והפעל את היישום. בטופס Product Details אשר יופיע, הקלד Aniseed Syrup בתיבת הטקסט Product Name ולחץ Get Product.

לאחר השהיה קצרה, בה יישום הלקוח יוצר מופע של המתווך (proxy), זה ממיר את הפרמטרים ושולח את הבקשה לשירות. השירות קורא את מסד הנתונים, יוצר אובייקט מסוג **Product**, ממיר אותו ל-XML, ושולח אותו בחזרה למתווך. המתווך ממיר את הנתונים בחזרה, יוצר עותק של האובייקט **Product** ומעביר את העותק לקוד שבשיטה **getButtonClick**. כעת יופיעו נתוני המוצר Aniseed Syrup על גבי הטופס כפי שמוצג במסך הבא:

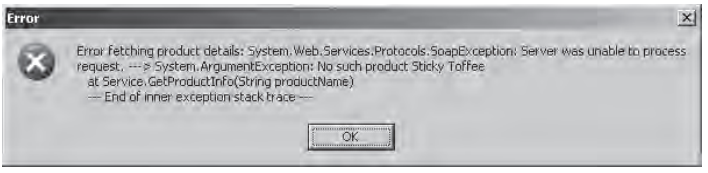


2. הקלד **Tofu** בתיבת הטקסט Product Name ולחץ Get Product.

כנראה שהפעם הנתונים יופיעו מהר יותר.

3. הקלד Sticky Toffee בתיבת הטקסט Product Name ולחץ Get Product פעם נוספת.

מכיוון שמוצר זה אינו קיים, שירות הרשת יזרוק חריג המועבר בחזרה ליישום. אם תביט בהודעה מקרוב תבחין בשורה "No such product Sticky Toffee".

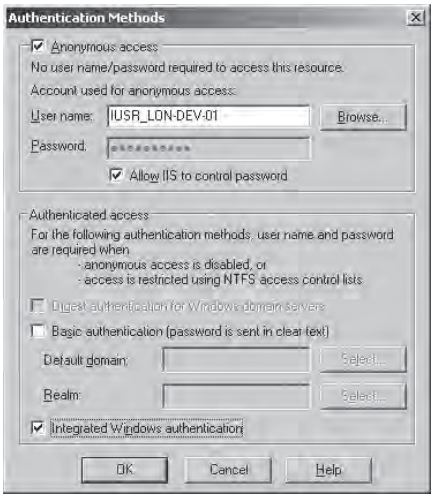


4. לחץ OK כדי לסגור את הודעת השגיאה. סגור את הטופס Product Details וחזור לסביבת התכנות של Visual Studio 2005.

שירותי רשת, גישה אנונימית ואימות

בעת יצירת לקוח של שירות רשת באמצעות Visual Studio 2005, יישום הלקוח, על פי ברירת המחדל, מפעיל את שירותי הרשת על ידי גישה אנונימית. למרות שאין עם זה בעיה בעת בנייה ובדיקה של שירותי רשת בסביבת הפיתוח ובאמצעות השרת Development Web server של ASP.NET, ייתכן שבסביבת הייצור (production environment) של השרת IIS תרצה להגביל את הגישה למשתמשים שזהותם אומתה בלבד (נכון לתרגילים אלו השירות אינו נמצא ב-IIS).

כדי לקבוע את תצורת מנגנוני האימות של שירות הרשת הממוקם בשרת IIS, עליך להשתמש במנגנון הבקרה של Internet Information Services הנמצא בתיקייה Administrative Tools אשר בלוח הבקרה (Control Panel). כדי לשנות את הגדרות Directory Security הרחב את הרכיב Default Web Site, סמן את שירות הרשת שלך, בתפריט Action בחר Properties ולבסוף לחץ על הלחצן Edit אשר נמצא תחת הסעיף Anonymous Access And Authentication Control. כעת ודא שתיבת הסימון Anonymous Access אכן מסומנת, כדי לאפשר גישה ללקוחות שזהותם אינה מאומתת. גישה כזו מתבצעת לרוב באמצעות חשבון IUSR מקומי. שירות הרשת יופעל באמצעות זהות זו, אשר מספקת גישה לכל המשאבים המשמשים את השירות (מסד נתונים של SQL Server, למשל).



החלופה לשימוש בגישה אנונימית היא שימוש בגישה מאומתת. אם תבטל את סימון התיבה Anonymous Access ותבחר באחד ממנגנוני האימות (Digest, Basic, Integrated Windows) או יותר, תוכל להגביל את הגישה רק עבור משתמשים בעלי שם משתמש וסיסמה תקפים. למידע נוסף על הבדלים בין שיטות האימות השונות עיין ב- Internet Information Services documentation (בתיבת הדו-שיח Authentication Methods לחץ Help).

את המידע הדרוש לאימות זהות המשתמש עליך להעביר, בסביבת ההרצה, דרך המאפיין **Credentials** של המתווך (proxy) של שירות הרשת. מאפיין זה הוא אובייקט **NetworkCredential** (המחלקה **NetworkCredential** נמצאת במרחב השמות **System.Net**). הקוד שלהלן יוצר אובייקט **NetworkCredential** עבור המשתמש "John" אשר סיסמתו היא "JohnPassword", וקובע את ערך המאפיין **Credentials** של האובייקט המתווך של השירות:

```
using System.Net;
…
private void getProduct_Click(...)
{
   Service northwindService = new Service();
   try
{
      NetworkCredential credentials =
         new NetworkCredential("John", "JohnsPassword");
      productService.Credentials = credentials;
      Product prod = productService.GetProductInfo(...);
      …
   }
   …
}
```

שם המשתמש והסיסמה חייבים להיות תקפים במסגרת ה-domain של Windows (ניתן להוסיף לבנאי **NetworkCredential** פרמטר שלישי המפרט תחום Windows נוסף), ולחשבון זה חייבת להיות גישה (הרשאה) לכל המשאבים השונים המשמשים את שירות הרשת כדי שיפעל כשורה.

פרק 28 – טבלה מסכמת

המשימה	צריך
ליצור שירות רשת.	להשתמש בתבנית ASP.NET Web Service. באמצעות התכונה **WebService** פרט את מרחבי השמות המשמשים לזיהוי השירות. סמן את השיטות שברצונך לחשוף באמצעות התכונה **WebMethod**.
לבדוק שירות רשת.	לחץ לחיצה ימנית על הקובץ asmx שב- **Solution Explorer** ובתפריט הקיצור בחר View in Browser. חלון Internet Explorer יופעל, יעבור לכתובת URL של שירות הרשת ויציג את דף הבדיקה. לחץ על הקישור לשיטת הרשת שברצונך להפעיל. בדף הבדיקה של שיטת הרשת, הקלד ערך עבור כל אחד מהפרמטרים ולחץ Invoke. שיטת הרשת תופעל ותחולל מענה SOAP בתבנית XML שיוצג בחלון Internet Explorer.
להעביר נתונים מורכבים כפרמטרים וערכים מוחזרים של השיטות.	הגדר מחלקה להכלת הנתונים. ודא שיש גישה לכל אחד מהנתונים בתור שדות ציבוריים או בעזרת מאפיין ציבורי הכולל אחראי גישה (accessors) **get** ו-**set**. ודא שהמחלקה כוללת בנאי ברירת מחדל (שעשוי גם להיות ריק).
להוסיף ליישום הפניה וליצור מחלקה מתווכת (proxy class).	בתפריט Project בחר Add Web Reference. הקלד בתיבת הטקסט Address את הכתובת URL של שירות הרשת או לחילופין לחץ על הקישור "Web Reference on local Web server" ובחר בשירות הרשת הרצוי. לחץ Add Reference כדי ליצור את המתווך (proxy) של השירות.
הפעל שיטת רשת.	צור מופע של המחלקה המתווכת. המחלקה נמצאת במרחב שמות ששמו זהה לזה של שרת הרשת המארח (host) את השירות, אלא אם כן תשנה אותו בעת הוספת הפניית הרשת. הפעל את השיטה באמצעות המחלקה המתווכת.

חלק ז'

C# 3.0

בחלק זה:

פרק 29

מה חדש ב- C# 3.0

מאז הפיתוח של שפת #C בשלהי שנת 2002, קיבלה מיקרוסופט משובים רבים מאנשי פיתוח ברחבי העולם בדבר שינויים ושיפורים בשפה.

אחד הדברים החיוביים במיקרוסופט הוא שאנשיה יודעים להקשיב, לשפר ולשנות. ואכן, הגרסה האחרונה והמלהיבה של C# 3.0 עתירה בשיפורים ושינויים (פנימיים). יתרה מכך, אנשי מיקרוסופט לקחו קטעי קוד שלמים ושכיחים, כפי שיוצגו מייד, פישטו ותמצתו את צורת הכתיבה שלהם, כדי שעבודת המתכנת תהיה פשוטה, ברורה ויעילה הרבה יותר.

התוספות שנראה מייד הן אבני דרך לדבר האמיתי - **LINQ**, אשר נסקור בפרק הבא.

מאפיינים אוטומטיים – Automatic Properties

בפרק 14 בסעיף "מהם מאפיינים?" למדת על **מאפיינים**.

נבחן את דוגמת הקוד הבאה:

```
public class Customer
{
    private int _customerID;
    private string _customerName;

    public int CustomerID{
        get
        {
            return _customerID;
        }
        set
        {
            _customerID = value;
        }
    }

    public string CustomerName
    {
        get
        {
            return _customerName;
        }
        set
        {
            _customerName = value;
        }
    }
    ...
}
```

קוד זה מכיל הגדרה של מחלקה פשוטה של לקוח, ושני מאפיינים: קוד לקוח ושם לקוח. שים לב שאין לוגיקה המופעלת בזמן גישה או השמה למאפיינים.

בדיוק עבור מצבים אלה, C# 3.0 מאפשרת להשתמש **במאפיינים אוטומטיים**.

הקוד הבא זהה בדיוק לקודמו, אלא שהוא קצר וקריא יותר:

```
public class Customer
{
    public int CustomerID { get; set; }
    public string CustomerName { get; set; }
    ...
}
```

במקום להגדיר משתנה פרטי ולאחר מכן להגדיר לו מאפיינים באופן ידני, המהדר עושה זאת מאחורי הקלעים.

יהיו מי שישאלו: אם כך, למה לא להשתמש בשדה (משתנה) ציבורי במקום מאפיינים ריקים?

יש סיבות רבות להעדפת השימוש במאפיינים, והסיבה העיקרית היא: כאשר אני משתמש במאפיינים (אפילו ריקים), אני יכול להוסיף להם לוגיקה, ולא לפגוע ב"חוזה" של המחלקה שלי עם קבצים שמשתמשים בה. כל שאני צריך לעשות הוא לערוך שינוי במאפיינים כדי להדר את המחלקה, ושאר הקבצים שפעלו עם המחלקה שלי אפילו לא יבחינו בשינוי (כימוס נתונים).

ניתן להשתמש גם במאפיינים אוטומטיים כאשר רוצים להשתמש במאפיין 'קריאה בלבד'. במקרה כזה ניתן להגדיר את מאפיין הגישה **set** לקריאה בלבד, כך:

```
public class Customer
{
    public int CustomerID { get; private set; }
    public string CustomerName { get; private set; }
    ...
}
```

אתחול אובייקטים – Object Initializers

מאתחלי אובייקט מפשטים את תהליך בניית האובייקט בכך שהם מאפשרים לקבוע מאפיינים ברצף לאחר הקריאה לבנאי. כתוצאה, במקום הקוד הבא:

```
Customer newCustomer = new Customer();
newCustomer.CustomerID = 1;
newCustomer.CustomerName = "Armika.ltd";
newCustomer.City = "Tel Aviv";
```

המהדר של C# 3.0 יאפשר לך לכתוב את הקוד בדרך זו:

```
Customer newCustomer = new Customer() { CustomerID = 2,
                                        CustomerName = "Armika.ltd",
                                        City = "Tel Aviv" };
```

מייד לאחר הגדרת האובייקט אני יכול להגדיר בתוך סוגריים מסולסלים את המאפיינים של האובייקט, והמהדר יפעיל באופן אוטומטי את מאפייני ההשמה של האובייקט ויעביר לו נתונים.

ניתן גם לאתחל מחלקה אשר מקבלת ערך בבנאי שלה בדרך זו:

```
Customer newCustomer = new Customer(2) { CustomerName = "Armika.ltd",
                                         City = "Tel Aviv" };
```

אתחול אוספים – Collection Initializers

בדוגמה הבאה אני משתמש באתחול אובייקטים למילוי אוסף של לקוחות:

```
List<Customer> Customers = new List<Customer>();

Customers.Add(new Customer(){CustomerID=1,CustomerName ="Hod Ami"});
Customers.Add(new Customer() { CustomerID = 2, CustomerName = "Matrix" });
Customers.Add(new Customer() { CustomerID = 3, CustomerName = "Leumi" });
```

בדומה לאתחול אובייקטים, C# 3.0 מאפשר לי לאתחל אוספים, למעשה כל אובייקט שיממש את הממשק **IEnumerable**, ובכך לחסוך לי גם את ההקלדה של השיטה Add:

```
List<Customer> Customers = new List<Customer>()
{
   new Customer() { CustomerID=1,CustomerName ="Hod Ami" },
   new Customer() { CustomerID = 2, CustomerName = "Matrix" },
   new Customer() { CustomerID = 3, CustomerName = "Leumi" }
};
```

כאשר המהדר מזהה את השורות מעל, הוא יחולל באופן אוטומטי את שיטות ההוספה של האוסף, ממש כמו בקוד הקודם.

משתנה מקומי מוגדר –

Implicitly typed local variable (var)

משתנה מקומי מוגדר הינו משתנה מקומי שמוגדר **ללא** ציון סוג המשתנה באופן מפורש.

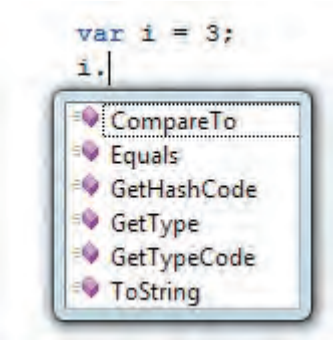
הבט בדוגמת הקוד הבאה:

```
int i = 3;
Customer newCustomer = new Customer();
```

כאן מוגדר משתנה מסוג **int** אשר מאותחל בערך 3. הערך 3 מייצג ללא ספק משתנה מסוג int. הגדרתי משתנה (אובייקט) נוסף בשם **newCustomer** מסוג המחלקה **Customer**. המהדר של C# 3.0 מאפשר להשמיט את סוג המשתנה במשפט ההכרזה ולהשתמש בפקודה **var**. פעולה זו מאפשרת למהדר לקבוע את סוג המשתנה על ידי צד ימין של הביטוי בזמן ההכרזה. מכיוון שהמהדר יכול לקבוע את סוג המשתנים, נוכל לכתוב את הקוד הבא:

```
var i = 3;
var newCustomer = new Customer();
```

כאשר הגדרתי את i באמצעות הפקודה var, המהדר מתייחס ל-i כמשתנה מסוג int, וכאשר אקליד (.) לאחר המשתנה i, אקבל את רשימת השיטות של המחלקה int:



השיטה **GetType()** של i תחזיר את הסוג System.Int32.

ניתן לראות את var כשומר מקום, אשר יוחלף על ידי המהדר בסוג אמיתי על פי סוג ערך ההשמה.

בדרך זו אפשר להגדיר משתנים מסוגים שונים:

```
var i = 3;
var s = "Hello";
var d = 1.0;
var numbers = new int[] { 1, 2, 3};
var newCustomer = new Customer();
```

יש מספר הגבלות על השימוש במשתנה מקומי מוגדר:

1. המשתנה חייב להיות מקומי.
2. חובה לאתחל משתנה. שלא כמו משתנה רגיל, אי אפשר להגדיר משתנה var מבלי לאתחל את המשתנה באותה פקודה.
3. אין אפשרות לאתחל משתנה בערך null.
4. אין אפשרות להחליף בין הסוגים השונים לאחר הגדרת משתנה מסוג מסוים.
5. אין אפשרות להגדיר מערכים מסוג var.

```
// ERROR: Implicitly-typed local variables must be initialized
var r;

// ERROR: Cannot implicitly convert type 'int[]' to 'var[]'
var[] arr = new int[] { 10, 20, 30 };

// ERROR: Cannot assign <null> to an implicitly-typed local variable
var n = null;

// ERROR: Cannot implicitly convert type 'string' to 'int'
var i = 3;
i = "you are int...";
```

שיטות הרחבה – Extension methods

שיטות הרחבה מאפשרות להוסיף תפקודיות למחלקות קיימות על ידי הוספת שיטות חדשות למחלקות קיימות, ללא צורך ליצור מחלקה נגזרת או לעדכן מחלקה קיימת. יש גם מצבים שאין אפשרות להוסיף פונקציונליות, כי אין את קוד המקור, ולפעמים אין אפשרות לרשת מחלקה מפני שהיא חתומה.

שיטות אלו הן סוג מיוחד של שיטות סטטיות, אך הקריאה להן נעשית באמצעות מופע של המחלקה שברצונך להרחיב. למעשה, אין הבדל בין פנייה לשיטות הרגילות של המחלקה לבין פנייה לשיטות מורחבות.

אם היית צריך לבצע בדיקת תקינות לכתובת דואר אלקטרוני, היית כותב קוד כזה, למשל:

```
string email = "dev@krudo.net";

if ( IsValidEmailAddress(email) )
{
   // Valid
}
else
{
   // email address is not valid
}
```

הפונקציה **IsValidateEmailAddress** תחזיר "שקר" או "אמת" על פי התקינות של כתובת הדואר האלקטרוני שהועברה כפרמטר.

בעזרת שיטות הרחבה ניתן להרחיב את התפקודיות של המחלקה **string** על ידי "הוספה" של השיטה **IsValidateEmailAddress** למחלקה, וכך זה ייראה:

```
var email = "dev@krudo.net";

if( email.IsValidEmailAddress() )
{
   // Valid
}
else
{
   // email address is not valid
}
```

ראוי לשים לב שכעת השיטה IsValidateEmailAddress מופעלת כאילו הייתה שיטה בתוך המחלקה string. הדרך להוסיף שיטת הרחבה היא להוסיף מחלקה סטטית שמכילה שיטה סטטית, אשר בשילוב מילת המפתח **this** היא מקבלת כפרמטר את סוג המחלקה שרוצים להרחיב:

```
public static class stringUtils
{
  public static bool IsValidEmailAddress(this string emailAddress)
  {
     Regex regex = new Regex(@"^[\w-\.]+@([\w-]+\.)+[\w-]{2,4}$");
     return regex.IsMatch(emailAddress);
  }
}
```

השילוב של מילת המפתח this מורה למהדר להוסיף את השיטה המורחבת למחלקה string.

דוגמה נפוצה נוספת היא הרחבת התפקודיות של המחלקה string, באופן ששיטה מורחבת של המחלקה תחזיר את מלל המשתנה בסדר הפוך:

```
public static class stringUtils
{
    public static string Reverse(this string input)
    {

        char[] inputArray = input.ToCharArray();
        Array.Reverse(inputArray);

        return new string(inputArray);

    }
 }
```

```
class Program
{
    static void Main(string[] args)
    {
        var str = "C# 3.0 is Cool!!!";
        Console.WriteLine( Str.Reverse() );
    }
}
```

סוגים אנונימיים – Anonymous Types

עד לגרסה C# 3.0, לצורך הגדרת משתנה (אובייקט) מסוג מחלקה כלשהי, צריך היה לייצר את המחלקה ורק אחר כך ליצור מופע (אובייקט) של אותה מחלקה.

זה היה נראה בערך כך:

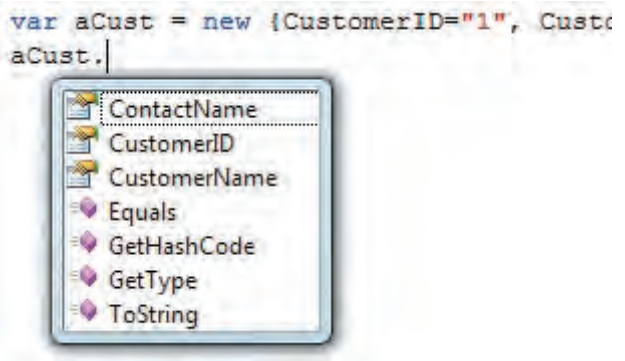
```
public class Customer
{
    public int CustomerID { get; set; }
    public string CustomerName { get; set; }
    public string ContactName { get; set; }
    public string City { get; set; }
}

class Program
{
    static void Main(string[] args)
    {
        Customer aCust = new Customer();
        ...

    }
}
```

ואז הגיעה C# 3.0 והוסיפה לארסנל את **הסוג האנונימי** (Anonymous Types), שמאפשר (בעזרת שני המאפיינים '**משתנה מקומי מוגדר**' ו'**אתחול אובייקטים**') להגדיר משתנה אובייקט מבלי להגדיר את סוגו.

'סוג אנונימי' מאפשר לכתוב את הקוד הקודם בדרך זו:

```
var aCust = new {CustomerID="1", CustomerName="Armika.ltd",
                 ContactName="Jacob"};
```

כבונוס, נקבל רשימת שיטות ומאפיינים:



אם אתה שואל את עצמך: "מהיכן מגיעה רשימת השיטות והמאפיינים?", ובכן כאשר המהדר של C# 3.0 מזהה הגדרה של סוג אנונימי, הוא מייצר (לשימושו הפרטי) מחלקה כמו זו:

```
class __Anonymous1
{
    private string _customerID = "1";
    private string _customerName = "Armika.ltd";
    private string _contactName = "Jacob";

    public string CustomerID
    {
        get { return _customerID; }
        set { _customerID = value; }
    }

    public string CustomerName
    {
        get { return _customerName; }
        set { _customerName = value; }
    }

    public string ContactName
    {
        get { return _contactName; }
        set { _contactName = value; }
    }
}
```

דרך מחלקה זו, המהדר מציג את מבנה הסוג האנונימי.

את החשיבות של סוגים אנונימיים וכיצד הם באים לידי ביטוי נראה בפרק הבא שיעסוק ב-LINQ.

ביטוי למבדא – Lambda Expression

בפרק 16 הכרנו את **השיטות האנונימיות**, אשר אפשרו לכלול קטעים קצרים של קוד ברצף בעת יצירה או הוספה של נציגים. אולם, התחביר המשמש לכתיבת שיטות אנונימיות מרובה במילים ומחייב שימוש במידע שהמהדר עצמו כבר מכיר. כדי לחסוך הקלדת נתונים שלא לצורך, C# 3.0 כוללת את האפשרות **ביטוי למבדא** כדי לקצר את התחביר של שיטות אנונימיות. סביר להניח שתרצה להשתמש בביטוי למבדא כתחליף לשיטות אנונימיות, ולמעשה, אם ביטויי למבדא היו מוצגים בגרסאות קודמות, לא היה כלל צורך בשיטות אנונימיות.

בתחביר של שיטה אנונימית, מילת המפתח **delegate** מיותרת, מכיוון שהמהדר כבר יודע שהשיטה מוקצית לנציג. ניתן להמיר שיטה אנונימית לביטוי למבדא בקלות רבה על ידי ביצוע הפעולות הבאות:

- מחיקת מילת המפתח delegate.
- מיקום אופרטור למבדא, =>, בין רשימת הפרמטרים לבין גוף השיטה האנונימית.

סימן אופרטור למבדא מבוטא כ"הולך אל".

הקוד שלהלן מציג את תהליך ההמרה. השורה הראשונה מציגה הקצאת שיטה אנונימית למשתנה **del**. השורה השנייה מציגה את אותה השיטה לאחר המרתה לביטוי למבדא, והקצאתה למשתנה **le**.

```
MyDel del = delegate(int x)    { return x + 1; }; // Anonymous method
MyDel le  =         (int x) => { return x + 1; }; // Lambda Expression
```

הערה

המונח **ביטוי למבדא** מבוסס על המונח **תחשיב למבדא** (lambda calculus), אשר פותח בשנות ה-20 וה-30 של המאה העשרים, על ידי אלונזו צ'רץ' ומתמטיקאים אחרים. תחשיב למבדא הינו מערכת לייצוג פונקציות, אשר עושה שימוש באות היוונית למבדא (λ) לייצוג פונקציה חסרת שם. לאחרונה, שפות תכנות פונקציונליות כדוגמה Lisp והדיאלקטים השונים שלה, עושות שימוש במונח לייצוג ביטויים שבהם ניתן להשתמש לתיאור ישיר של הגדרת פונקציה, במקום שימוש בשם ייחודי.

ההמרה הפשוטה משתמשת בפחות מילים ונראית "נקייה" יותר, אך בשלב זה היא חוסכת למעשה ששה תווים. אולם, יש מידע נוסף שהמהדר יכול לגזור באופן אוטומטי, ולכן תוכל להמשיך ולפשט את ביטוי הלמבדא, כמוצג בקוד שלהלן:

- בהסתמך על הכרזת הנציג, המהדר יכול לגזור את סוגי הפרמטרים של הנציג. לכן, השימוש בביטוי למבדא מאפשר להשמיט את סוגי הפרמטרים, כמוצג בהקצאה ל-le2.
  - פרמטרים שמופיעים בצירוף הסוג שלהם, מכונים פרמטרים **בסיווג מפורש**.
  - פרמטרים שמופיעים ללא הסוג שלהם, מכונים פרמטרים **בסיווג מרומז**.

- כאשר יש פרמטר יחיד בסיווג מפורש, תוכל להשמיט את הסוגריים שתוחמים אותו, כמוצג בהקצאה ל-le3.
- לסיום, ביטויי למבדא מאפשרים לגוף הביטוי להיות בלוק משפט או ביטוי. כאשר בלוק המשפט מכיל משפט החזרה יחיד, תוכל להחליף את בלוק המשפט עם הביטוי שמופיע לאחר מילת המפתח **return**, כמוצג בהקצאה ל-le4.

```
MyDel del = delegate(int x)    { return x + 1; };  // Anonymous method
MyDel le1 =         (int x) => { return x + 1; };  // Lambda Expression
MyDel le2 =             (x) => { return x + 1; };  // Lambda Expression
MyDel le3 =               x => { return x + 1; };  // Lambda Expression
MyDel le4 =               x =>          x + 1   ;  // Lambda Expression
```

התצורה הסופית של ביטוי למבדא מכילה כרבע ממספר התווים שבשיטה האנונימית המקורית, והיא הרבה יותר "נקייה" וקלה להבנה.

הקוד הבא מציג את תהליך ההמרה בשלמותו. השורה הראשונה של **Main** מציגה שיטה אנונימית והקצאתה למשתנה **del**. השורה השנייה מציגה את אותה השיטה לאחר המרתה לביטוי למבדא והקצאתה למשתנה le1.

```
delegate double MyDel(int par);

static void Main(string[] args)
{
  MyDel del = delegate(int x) { return x + 1; };
                                                    // Anonymous method

  MyDel le1 = (int x) =>   { return x + 1; };
                                                    // Lambda expression
  MyDel le2 =     (x) =>     { return x + 1; };
  MyDel le3 =      x =>      { return x + 1; };
  MyDel le4 =      x =>               x + 1;

  Console.WriteLine("{0}", del(12));
  Console.WriteLine("{0}", le1(12)); Console.WriteLine("{0}", le2(12));
  Console.WriteLine("{0}", le3(12)); Console.WriteLine("{0}", le4(12));
}
```

להלן מספר נקודות חשובות בדבר רשימת הפרמטרים של ביטוי למבדא:

- הפרמטרים ברשימת הפרמטרים של ביטוי למבדא חייבים להתאים לאלה של הנציג במספרם, סוגם ומיקומם.
- הפרמטרים ברשימת הפרמטרים של ביטוי למבדא אינם חייבים לכלול את הסוג (סיווג מרומז) אלא אם הנציג מכיל פרמטרים מסוג **out** או **ref** - במקרה שכזה חובה לציין את הסוג (סיווג מפורש).
- כאשר יש פרמטר אחד בלבד והסיווג שלו מרומז, ניתן להשמיט את הסוגריים התוחמים. כאשר יש יותר מפרמטר אחד, חובה להשתמש בהם.
- כאשר אין פרמטרים כלל, חובה להשתמש בצמד סוגריים ריקים.

סיכום

כפי שראינו, חברי צוות #C של מיקרוסופט היו עסוקים בשדרוג השפה. כל השיפורים שנוספו לשפה נועדו בסופו של דבר לתמוך ב-**LINQ**, אך גם מי שלא יעבוד עם LINQ בהחלט ייהנה מחוויית פיתוח.

באמצעות אתחול האוספים והאובייקטים ניתן לכתוב פחות קוד ליצירת אובייקטים.

השימוש במשתנה מקומי מוגדר ומשתנים אנונימיים מאפשר ליצור נתונים וסוגי נתונים באופן מיידי.

שיטות הרחבה מאפשרות להוסיף תפקודיות למחלקות חסומות או מחלקות שאין בידינו את קוד המקור שלהן (דבר שהיה בלתי אפשרי בעבר).

ביטוי למבדא (lambda expressions) מקצר את התחביר של שיטות אנונימיות. סביר להניח שכשרוצים להשתמש בביטוי למבדא כתחליף לשיטות אנונימיות, ולמעשה, אם ביטויי למבדא היו מוצגים מוקדם יותר, לא היה כלל צורך בשיטות אנונימיות.

פרק 30

מבוא ל–LINQ

LINQ הינה שפת שאילתות חדשה לשפות #C ו– Visual Basic.NET, אשר משלבת יכולות חקירת נתונים בשפות תכנות אלו.

במערכת בסיס נתונים טבלאי, הנתונים מאורגנים בטבלאות אשר הגישה אליהן מתבצעת באמצעות **שפת שאילתות** (**query language**) פשוטה, אך יעילה ביותר – **SQL**. שפת SQL יכולה לפעול עם כל מערך של נתונים בבסיס הנתונים, מכיוון שהנתונים מאורגנים בטבלאות על פי חוקים מפורשים.

אולם בתוכנית, בניגוד לבסיס נתונים, הנתונים מאוחסנים באובייקטים מחלקתיים, או מבנים שונים מאוד זה מזה. לכן, עד כה לא פותחה שפת שאילתות כללית למטרת שליפת נתונים ממבני נתונים. השיטות המשמשות לשליפת נתונים מאובייקט עוצבו תמיד בהתאמה אישית כחלק מהתוכנית. עם זאת, מאז הופעת LINQ ב– C# 3.0, התווספה לשפה היכולת לחקור אוספי אובייקטים. להלן כמה מהמאפיינים המרכזיים החשובים ביותר של LINQ:

- משמעות המונח LINQ (מבוטא "link") הינה **שאילתה בשפה אחידה**.
- LINQ הוא חלק מתוך Net Framework., והוא מאפשר לחקור אוספי נתונים בצורה דומה לאופן החקירה של בסיסי נתונים.
- שפת C# 3.0 כוללת הרחבות שמטרתן לשלב את יכולות LINQ בשפה, ואשר מאפשרות לחקור נתונים בבסיסי נתונים, באוספי אובייקטים של תוכניות ובמסמכי XML.

הקוד הבא מציג דוגמה פשוטה לשימוש ב–LINQ. בקוד זה, מקור הנתונים הנחקר הוא מערך של שלמים (**int**). הגדרת השאילתה מבוצעת במשפט עם מילות המפתח **from** ו–**select**. למרות שהשאילתה **מוגדרת** במשפט זה, בפועל היא מבוצעת ומיושמת רק אחר כך במשפט **foreach** שבתחתית הקוד.

```
static void Main(string[] args)
{
    // מקור הנתונים
    int[] numbers = { 11, 2, 3, 5, 1, 15, 6, 22 };

    // הגדרת ושמירת השאילתה במשתנה underTen
    IEnumerable<int> underTen =
                    from n in numbers
                    where n < 10
                    select n;

    // ביצוע (הפעלה) של השאילתה
    foreach (var x in underTen)
        Console.Write("{0}, ", x);
}
```

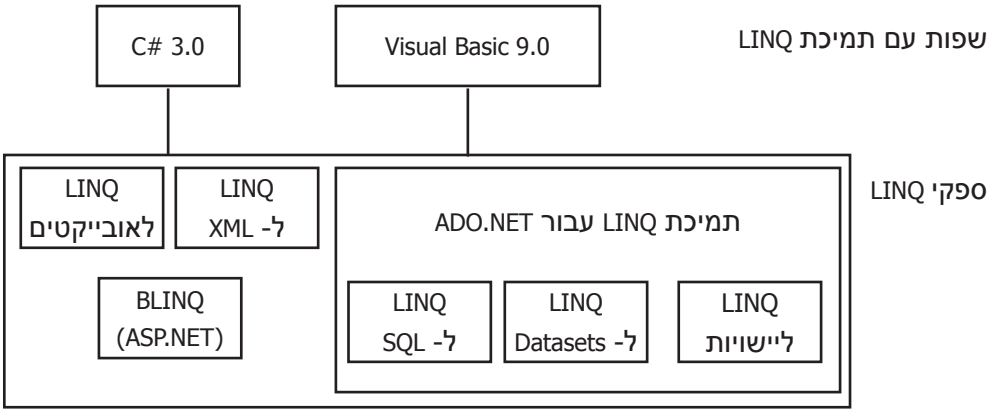
קוד זה יפיק את הפלט הבא:

```
2, 3, 5, 1, 6,
```

ספקי LINQ (LINQ Providers)

בדוגמה הקודמת, מקור הנתונים היה מערך פשוט של משתני int, המהווה למעשה אובייקט בזיכרון. אולם, שפת השאילתות LINQ יכולה לפעול עם סוגים רבים ומגוונים של מקורות נתונים, כמו למשל בסיסי נתונים SQL, מסמכי XML ועוד. אולם, עבור כל סוג של מקור נתונים, מאחורי הקלעים יש צורך במודול של קוד שמיישם את שאילתות LINQ במונחים שמתאימים לסוג מקור הנתונים. מודולי הקוד הללו נקראים **ספקי LINQ**. להלן מספר נקודות חשובות שצריך לדעת בנושא ספקי LINQ:

- Microsoft פיתחה ספקי LINQ עבור מספר סוגים של מקורות נתונים, כמוצג בתרשים 30-1.
- ניתן להשתמש בספקי LINQ בכל שפה עם תמיכת LINQ (במקרה שלנו, C# 3.0) לחקירת כל סוג של מקור נתונים אשר פותח עבורו ספק LINQ.
- ספקי LINQ חדשים מפותחים ללא הרף על ידי מספר גופים חיצוניים עבור סוגים שונים של מקורות נתונים.



תרשים 30-1: ארכיטקטורת LINQ, שפות עם תמיכת LINQ וספקי LINQ

תחביר שאילתות ותחביר שיטות

בעת כתיבת שאילתות LINQ תוכל להשתמש בשתי תצורות תחביר שונות - תחביר שאילתות ותחביר שיטות.

- **תחביר שאילתות** מהווה תצורה הצהרתית בעלת מראה דומה מאוד למשפטי SQL. תחביר שאילתות נכתב בתצורת **ביטוי שאילתה**.

- **תחביר שיטות** מהווה תצורת ציווי, אשר משתמש בקריאות מקובלות לשיטות. השיטות נלקחות מאוסף שמכונה **אופרטורי שאילתה סטנדרטיים** (standard query operators), אשר נתאר בהמשך הפרק.
- ניתן לשלב את שתי התצורות בשאילתה אחת.

תחביר השאילתות קריא יותר ומציין בצורה ברורה יותר את כוונת השאילתה, וככזה הוא מועד פחות לשגיאות. עם זאת, יש מספר אופרטורים שניתנים לכתיבה רק באמצעות תחביר שיטות.

הקוד שלהלן כולל את שלוש תצורות השאילתה השונות. אופן כתיבת הפרמטרים של השיטה **Where** משתמשת **בביטוי למבדא**, כפי שהוסבר בפרק הקודם.

```
static void Main(string[] args)
{
    int[] numbers = { 9, 2, 3, 4, 1, 15, 6, 22 };

    // תחביר שאילתה
    var pairNumbers1 = from n in numbers
                       where n % 2 == 0
                       select n;

    // תחביר שיטה
    var pairNumbers2 = numbers.Where(x => x % 2 == 0);

    // תחביר משולב
    int pairNumbersCount = (from n in numbers
                            where n % 2 == 0
                            select n).Count();

    foreach (var x in pairNumbers1)
        Console.Write("{0}, ", x);
    Console.WriteLine();


    foreach (var x in pairNumbers2)
        Console.Write("{0}, ", x);
    Console.WriteLine();

    Console.WriteLine(pairNumbersCount);
}
```

קוד זה יפיק את הפלט הבא:

```
2, 4, 6, 22,
2, 4, 6, 22,
4
```

משתני שאילתה

שאילתות LINQ יכולות להחזיר שני סוגי תוצאות: רשימה, אשר מונה את כל הפריטים שתואמים לפרמטרים של השאילתה; או ערך בודד, אשר ברוב המקרים מסכם את התוצאות שמקיימות את דרישות השאילתה.

לדוגמה, המשפט הראשון בקוד שלהלן מחזיר אובייקט בעל ממשק **IEnumerable**, אשר יכול לשמש לספירת תוצאות השאילתה. המשפט השני מוציא לפועל את השאילתה ואחר כך קורא לשיטה **Count** שמחזירה את ספירת הפריטים שהוחזרו על ידי השאילתה. בהמשך פרק זה נדון באופרטורים, כמו למשל Count, אשר מחזירים ערך בודד.

```
int[] numbers = { 1, 11, 21 };

// מחזיר רשימה
IEnumerable<int> underTenNums = from n in numbers
                                where n < 10
                                select n;

// מחזיר ערך בודד
int underTenCount = (from n in numbers
                     where n < 10
                     select n).Count();
```

המשתנה שמשמאל לסימן שווה מכונה **משתנה השאילתה**. למרות שסוגי המשתנים של השאילתה מוגדרים באופן מפורש במשפטים הקודמים, ניתן גם היה לאפשר למהדר להסיק את סוגי המשתנים של השאילתה על ידי החלפת שמות הסוגים עם מילת המפתח **var**.

חשוב להבין את תוכן משתני השאילתה. לאחר הפעלת הקוד המקדים, משתנה השאילתה **underTenNums** אינו מכיל עוד את תוצאות השאילתה, אלא אובייקט מסוג **IEnumerable<int>**, אשר יכול להפעיל את השאילתה אם ייקרא לעשות זאת בשלב מאוחר יותר של הקוד. משתנה השאילתה **underTenCount** מכיל ערך שלם ממשי שהשגתו יכולה להתבצע רק על ידי הרצה בפועל של השאילתה.

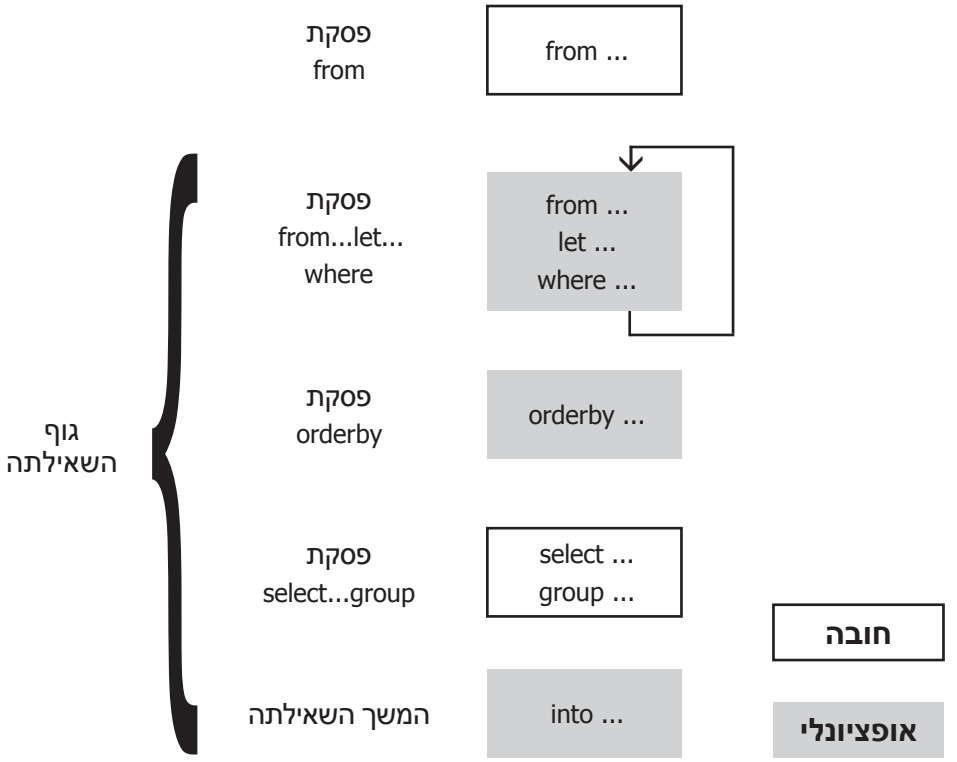
את ההבדלים בתזמון הפעלת השאילתות ניתן לסכם באופן הבא:

- כאשר השאילתה מחזירה **רשימה** (**enumeration**), הפעלת השאילתה תבוצע רק לאחר עיבוד הרשימה. אם הרשימה מעובדת מספר פעמים, השאילתה תופעל גם היא מספר פעמים.
- כאשר ביטוי השאילתה מחזיר ערך מסוג **סקלר** (**scalar**), הפעלת השאילתה מבוצעת באופן מיידי, והתוצאה מאוחסנת במשתנה השאילתה.

המבנה של ביטויי שאילתה

ביטוי שאילתה מורכב מפסקת **from** שאחריה גוף השאילתה, כמוצג בתרשים 2-30. להלן מספר נקודות חשובות שיש לדעת על ביטויי שאילתה:

- הפסקאות צריכות להופיע בסדר המוצג.
  - שני החלקים שחייבים להופיע הם פסקת **from** ופסקת **select...group**.
  - הפסקאות האחרות אינן מחייבות.
- בביטויי שאילתת LINQ, פסקת **select** צריכה להיות בסוף הביטוי. הדבר שונה משפת SQL, שבה משפט select מופיע בראש השאילתה. אחת הסיבות לשינוי זה ב-C# היא הרצון לאפשר לכלי **IntelliSense** של Visual Studio להציג אפשרויות נוספות במהלך הקלדת הקוד.
- תוכל לצרף מספר בלתי מוגבל של מקטעי **from...let...where**, כמוצג בתרשים שלהלן:



תרשים 30-2: המבנה של משפט שאילתה מורכב מפסקת from שאחריה גוף השאילתה.

הפסקה from

פסקת from מגדירה את אוסף הנתונים אשר יתפקד בתור מקור הנתונים. בנוסף לכך היא מציגה את משתנה האיטרציה. להלן מספר נקודות חשובות שעליך לדעת בנוגע לפסקת from:

- **משתנה האיטרציה** מייצג באופן רציף כל אחד מהאלמנטים שבמקור הנתונים.

- להלן מוצג התחביר של פסקת from, כאשר
  - **Type** הינו סוג האלמנטים שבאוסף. אינך חייב לציין את הסוג, מכיוון שהמהדר יכול להסיק מהו הסוג על פי האוסף.
  - **Item** הוא השם של **משתנה האיטרציה**.
  - **Items** הוא השם של האוסף שהשאילתה תופנה אליו. האוסף צריך להיות בעל ממשק IEnumerable.

from *Type Item* in *Items*

הקוד שלהלן מציג ביטוי שאילתה אשר משמש לחקירת מערך שכולל ארבעה ערכים שלמים (int). משתנה האיטרציה **item** מייצג כל אחד מארבעת הפריטים במערך, ובכל חזרה ייבחר או יידחה על ידי פסקאות **where** ו-**select** שמתחתיו. בקוד זה מבוצעת השמטה של הגדרת הסוג (int) האופציונלית של משתנה האיטרציה.

```
int[] grades = { 100, 90, 55, 60, 82, 75 };

var goodGrades = from grade in grades   // grade – משתנה איטרציה
                 where grade > 75
                 select grade;

foreach (var grade in goodGrades)
    Console.Write("{0}, ", grade);
```

קוד זה יפיק את הפלט הבא:

```
100, 90, 82,
```

למרות הדמיון הרב שבין פסקאות **from** של LINQ לבין משפטי **foreach**, יש מספר הבדלים משמעותיים בין השניים:

- הגוף של משפט **foreach** מופעל בשלב שבו הוא מופיע בקוד. פסקת from אינה מפעילה דבר, אלא יוצרת אובייקט בעל ממשק IEnumerable שמאוחסן במשתנה השאילתה. ביצוע השאילתה עצמה עשוי להתרחש בשלב מאוחר יותר של הקוד ועשוי גם שלא להתבצע כלל.
- משפט foreach מציין את הפריטים באוסף שצריך להתייחס אליהם לפי הסדר, מהראשון לאחרון. פסקת from מציינת באופן הצהרתי שיש להתייחס לכל הפריטים באוסף, אך אינה מגדירה סדר מסוים כלשהו.

הפסקה join

הפסקה join של LINQ דומה מאוד לפסקת JOIN של SQL. אם התנסית בעבר בשימוש בצירופים של SQL, הרי שהצירופים של LINQ לא יחדשו לך דבר מבחינה רעיונית, מלבד העובדה שניתן להפעילם על אוספים של אובייקטים, בנוסף לטבלאות בסיסי נתונים. אם צירופים הינם מושג חדש עבורך, או אם אתה זקוק לרענון, סעיף זה יסייע לך להבין את הרעיון בצורה טובה יותר.

חשוב לדעת מספר דברים בנוגע לצירופים:

- **פעולת צירוף** מקבלת שני אוספים ויוצרת אוסף זמני חדש של אובייקטים, שבו כל אחד מהאובייקטים מכיל את כל השדות של האובייקט אשר מופיע בשני האוספים המקוריים.
- **צירופים** משמשים לאיחוד נתונים משני אוספים או יותר.

התחביר אשר משמש לביצוע צירוף מוצג בדוגמה הבאה. קוד זה משמש לצירוף של האוסף השני לאוסף שבפסקה הקודמת.

מילת מפתח מילת מפתח מילת מפתח מילת מפתח
↓ ↓ ↓ ↓
join *Identifier* in *Collection2* on *Field1* equals *Field2*
↑ ↑
ציון של אוסף נוסף וזיהוי לשם פנייה שדות ההשוואה בין שני האוספים

תרשים 30-3: התחביר המשמש לכתיבת פסקת join.

join Identifier **in** Collection2 **on** Field1 **equals** Field2

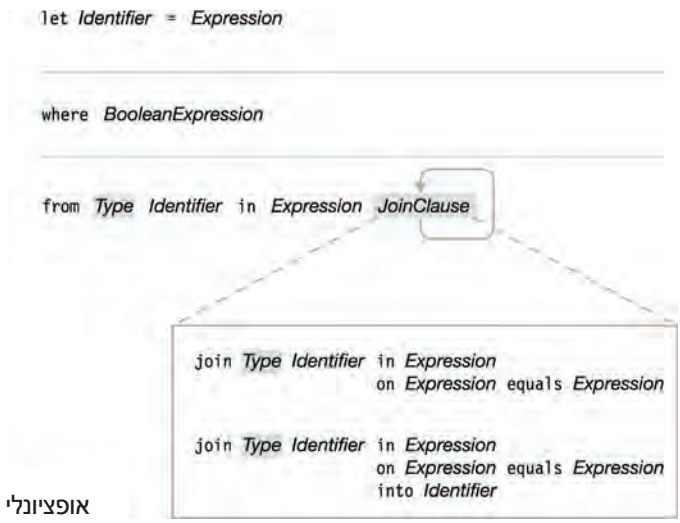
המילים המודגשות הן מילות מפתח, כאשר הביטוי: "Identifier in Collection2" מציין אוסף נוסף והביטוי: "Field1 equals Field2" מציין את השדות לזיהוי והשוואה.

דוגמה:

```
var commonProducts = from p in Products            // האוסף הראשון
                     join op in OrderedProducts    // האוסף השני
                     on p.ProductID equals op.ProductID
                                                   // השדות המקשרים להשוואה
```

מקטע from...let...where בגוף השאילתה

מקטע from...let...where האופציונלי הוא המקטע הראשון בגוף השאילתה. הוא יכול להכיל כל אחד משלוש הפסקאות המרכיבות אותו - פסקת **from**, פסקת **let** ופסקת **where**. התרשים הבא מסכם את התחביר המשמש לכתיבת שלוש הפסקאות (המלל המודגש הינו אופציונלי).



תרשים 4-30: תחביר פסקת from...let...where

הפסקה from

למדת שביטוי שאילתה חייב להתחיל בפסקת from, שאחריה גוף השאילתה. הגוף עצמו יכול להתחיל בכל מספר של פסקאות from נוספות, אשר כל אחת מהן משמשת לציון מקור נתונים נוסף ולהצגת משתנה איטרציה חדש לשימוש בהערכות עתידיות. התחביר והמשמעות זהים בכל פסקאות from.

הקוד שלהלן מדגים את השימוש בטכניקה זו.

- פסקת from הראשונה הינה הפסקה הנדרשת של ביטוי השאילתה.
- פסקת from השנייה הינה הפסקה הראשונה בגוף השאילתה.
- פסקת select יוצרת אובייקטים מסוג אנונימי.

```
static void Main(string[] args)
{
    var arr1 = new[] { 1, 3, 5 };
    var arr2 = new[] { 2, 4, 6 };

    var newArr = from a1 in arr1
                 from a2 in arr2
                 where a1 > 2 && a2 < 5
                 select new { a1, a2, sum = a1 + a2 };

    foreach (var a in newArr)
    {
        Console.WriteLine(a);
    }

}
```

קוד זה יפיק את הפלט הבא:

```
{ a1 = 3, a2 = 2, sum = 5 }
{ a1 = 3, a2 = 4, sum = 7 }
{ a1 = 5, a2 = 2, sum = 7 }
{ a1 = 5, a2 = 4, sum = 9 }
```

הפסקה let

הפסקה let מקבלת ערך של ביטוי ומקצה אותו למזהה (משתנה) לשימוש בהערכות אחרות. התחביר של פסקת let הוא:

```
let Identifier = Expression
```

לדוגמה, ביטוי השאילתה שבקוד הבא יוצר זוגות איברים שמורכבים מאלמנט במערך arr1 ואלמנט במערך arr2. הפסקה where מסלקת כל צמד של מספרים שלמים (integers) משני המערכים, אם סכום שני המספרים אינו שווה ל-10.

```
static void Main(string[] args)
{
    var arr1 = new[] { 1, 3, 5 };
    var arr2 = new[] { 2, 5, 7 };

    var newArr = from a1 in arr1
                 from a2 in arr2
                 let sum = a1 +a2  // אחסון התוצאות במשתנה חדש
                 where sum == 10
                 select new { a1, a2, sum };

    foreach (var a in newArr)
    {
        Console.WriteLine(a);
    }

}
```

קוד זה יפיק את הפלט הבא:

```
{ a1 = 3, a2 = 7, sum = 10 }
{ a1 = 5, a2 = 5, sum = 10 }
```

הפסקה where

הפסקה where שולפת פריטים מהתוצאה הסופית אם הם ממלאים תנאי מוגדר. התחביר של פסקת where הוא:

```
where BooleanExpression
```

להלן מספר נקודות חשובות שעליך לדעת בנוגע לפסקת where:

- ביטוי שאילתה יכול לכלול מספר בלתי מוגבל של פסקאות where, כל עוד הן נמצאות במסגרת מקטע from...let...where.
- פריט שאינו מקיים את הדרישות של כל הפסקאות where ינופה מתוצאת השאילתה.

הקוד מציג דוגמה לביטוי שאילתה אשר מכיל שתי פסקאות where. הפסקאות where מסננות כל צמד של שלמים משני המערכים אשר אינם מקיימים תנאי של סכום שני המספרים קטן מ-10, והאלמנט מ-arr1 אינו 5. כל צמד נבחר של אלמנטים חייב לקיים את התנאים של **שתי** פסקאות where.

```
static void Main(string[] args)
{
    var arr1 = new[] { 1, 3, 5 };
    var arr2 = new[] { 2, 5, 4 };

    var newArr = from a1 in arr1
                 from a2 in arr2
                 let sum = a1 +a2
                 where sum <= 10   // תנאי ראשון
                 where a1 == 5     // תנאי שני
                 select new { a1, a2, sum };

    foreach (var a in newArr)
    {
        Console.WriteLine(a);
    }

}
```

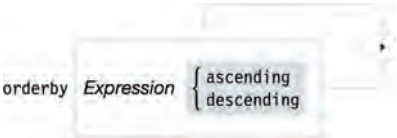
קוד זה יפיק את הפלט הבא:

```
{ a1 = 5, a2 = 2, sum = 7 }
{ a1 = 5, a2 = 5, sum = 10 }
{ a1 = 5, a2 = 4, sum = 9 }
```

הפסקה orderby

הפסקה orderby מקבלת ביטוי ומחזירה את פריטי התוצאה בסדר שמוגדר בו. התחביר של פסקת orderby מוצג בתרשים 5-30. מילות המפתח האופציונליות **ascending** (סדר עולה) ו-**descending** (סדר יורד) משמשות לקביעת כיוון הסידור. **Expression** בדרך כלל מהווה שדה בפריטים.

- על פי ברירת מחדל, פסקת orderby מחזירה את האלמנטים בסדר עולה. אולם, באפשרותך לקבוע באופן מפורש את כיוון הסידור של האלמנטים (עולה או יורד) באמצעות מילות המפתח ascending ו-descending.
- תוכל להוסיף מספר בלתי מוגבל של פסקאות orderby, אך עליך להפרידן באמצעות פסיקים.



תרשים 5-30: תחביר הפסקה orderby

הקוד שלהלן הינו דוגמה לרשומות של מוצרים אשר מסודרים לפי ערך המחיר. שים לב שנתוני המוצרים נמצאים במערך של סוגים אנונימיים.

```
static void Main(string[] args)
{
    var products = new[]  // מערך של סוג אנונימי
    {
        new {ProductID = 1, ProductName = "Bamba", Price=3.40d},
        new {ProductID = 2, ProductName = "Bisli", Price=3.60d},
        new {ProductID = 3, ProductName = "Beigale", Price=3.90d}
    };

    var sortProducts = from product in products
                       orderby product.Price  // מיון לפי מחיר
                       select product;

    foreach (var p in sortProducts)
    {
        Console.WriteLine("{0}. {1},   {2}"
                    , p.ProductID, p.ProductName, p.Price );
    }
}
```

קוד זה יפיק את הפלט הבא:

```
1. Bamba, 3.40
2. Bisli, 3.60
3. Beigale, 3.90
```

הפסקה select...group

שני סוגים של פסקאות מרכיבים את מקטע select...group – פסקת select ופסקת group...by. מקטע select...group משמש לביצוע הפעולות הבאות:

- הפסקה **select** מציינת איזה חלקים יישלפו מתוך האובייקט רצוי. היא יכולה לציין כל אחד מאלה:
  - את פריט הנתונים כולו
  - שדה בפריט הנתונים
  - אובייקט חדש שכולל מספר שדות מפריט הנתונים (או כל ערך אחר לצורך העניין).
- פסקת **group...by** אינה מחייבת, אלא משמשת לציון אופן ההקבצה של הפריטים הנבחרים. הסבר על הפסקה group...by ניתן בהמשך הפרק.

התחביר של הפסקה select...group:

```
select  Expression
-------------------------------------
group  Expression1  by  Expression2
```

תרשים 30-6: תחביר הפסקה select...group

הקוד הבא הינו דוגמה לשימוש בפסקה select לבחירת פריט הנתונים כולו. בשלב הראשון, מיוצר מערך של סוג אנונימי. לאחר מכן, ביטוי השאילתה משתמש במשפט select כדי לבחור כל אחד מהפריטים שבמערך.

```
static void Main(string[] args)
{
    var products = new[]  // מערך של סוג אנונימי
    {
        new {ProductID = 1, ProductName = "Bamba", Price=3.40d},
        new {ProductID = 2, ProductName = "Bisli", Price=3.60d},
        new {ProductID = 3, ProductName = "Beigale", Price=3.50d}
    };

    var sortProducts = from product in products
                       orderby product.Price  // מיון לפי מחיר
                       select product;

    foreach (var p in sortProducts)
    {
        Console.WriteLine("{0}. {1},   {2}"
                    , p.ProductID, p.ProductName, p.Price );
    }
}
```

קוד זה יפיק את הפלט הבא:

```
1. Bamba, 3.40
3. Beigale, 3.50
2. Bisli, 3.60
```

ניתן להשתמש בפסקה select גם כדי לבחור שדות ספציפיים מתוך האובייקט. לדוגמה, הפסקה select שבקוד שלהלן בוחרת רק את שם המוצר.

```
var Prod = from product in products
           select product.ProductName;

foreach (var p in Prod)
       Console.WriteLine(p);
```

לאחר שתחליף שני משפטים אלה עם שני המשפטים המקבילים שבדוגמה המלאה הקודמת, התוכנית תפיק את הפלט הבא:

```
Bamba
Bisli
Beigale
```

הפסקה group

הפסקה group מקבצת את האובייקטים הנבחרים בהתאם לקריטריון כלשהו מוגדר. לדוגמה, במערך המוצרים שבדוגמה הקודמת, נוכל לגרום לתוכנית לקבץ את המוצרים לפי סוגם.

להלן מספר נקודות חשובות שעליך לדעת על הפסקה group:

- כאשר יש פריטים בתוצאות השאילתה, הם מאורגנים בקבוצות בהתאם לערך של שדה מסוים. הערך שלפיו הפריטים מקובצים מכונה "**key**".
- בניגוד לפסקה select, הפסקה group אינה מחזירה אובייקט שמונה את הפריטים שבמקור באופן ישיר. במקום זאת, היא מחזירה אובייקט המונה את קבוצות הפריטים שנוצרו.
- הקבוצות יכולות למנות את הפריטים.

להלן דוגמה לתחביר אשר משמש לכתיבת הפסקה group:

```
group product by product.Category
```

לדוגמה, הקוד שלהלן מקבץ את המוצרים לפי קטגוריות:

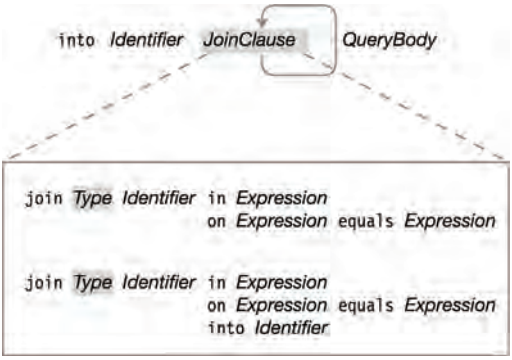
```
static void Main(string[] args)
{
 var products = new[]
 {
  new {ProductID = 1, ProductName = "milki",   Category = "Dairy"},
  new {ProductID = 2, ProductName = "Dani",    Category = "Dairy"},
  new {ProductID = 3, ProductName = "yogurt",  Category = "Dairy"},
  new {ProductID = 4, ProductName = "chicken", Category = "Meat"},
  new {ProductID = 5, ProductName = "steak",   Category = "Meat"}
 };

 var groupProducts = from product in products
                     group product by product.Category;

 foreach (var p in groupProducts)
 {
  Console.WriteLine("{0}",p.Key);

  foreach (var gp in p)
   Console.WriteLine("    {0}. {1}",gp.ProductID,gp.ProductName );

 }
}
```

קוד זה מפיק את הפלט הבא:

```
Dairy
1. milki
2. dani
3. yogurt
Meat
4. chicken
5.  steak
```

הפסקה הארכת שאילתה

הפסקה הארכת שאילתה מקבלת את התוצאה של אחת מפסקאות השאילתה ומקצה לה שם כדי שיהיה ניתן להשתמש בשם זה בחלקים אחרים של השאילתה. התחביר המשמש לכתיבת הארכת שאילתות מוצג בתרשים 7-30.



תרשים 7-30: תחביר הפסקה הארכת שאילתה

לדוגמה, השאילתה שלהלן מצרפת את arr1 ו-arr2 לתוך סוג חדש ומקצה לצירוף את השם arr1_2 (ביצוע פעולת הארכה) לאחר מכן היא מתשאלת את arr1_2.

```
static void Main(string[] args)
{
    var arr1 = new[] { 1, 3, 4 };
    var arr2 = new[] { 1, 4, 6 };

    var query = from a1 in arr1
                join a2 in arr2 on a1 equals a2
                into arr1_2        // כאן מתבצעת הארכת השאילתה
                from x in arr1_2
                select x;

    foreach (var a in query)
        Console.WriteLine("{0}", a);
}
```

קוד זה יפיק את הפלט הבא:

```
1
4
```

אופרטורי שאילתה סטנדרטיים

אופרטורי שאילתה סטנדרטיים מורכבים מסדרה של שיטות המאפשרות לחקור כל סוג של מערך או אוסף מסוג NET. להלן מספר מאפיינים חשובים של אופרטורי שאילתה סטנדרטיים:

- אובייקטי האוסף הנחקרים מכונים **רצפים** (**sequences**), והם חייבים ליישם את הממשק **IEnumerable<T>**, כאשר T הוא סוג.
- אופרטורי שאילתה סטנדרטיים משתמשים בתחביר שיטות.
- אופרטורים מסוימים מחזירים אובייקטים מסוג **IEnumerable** (או רצפים אחרים), בעוד שאופרטורים אחרים מחזירים ערך בודד. אופרטורים אשר מחזירים ערך בודד מפעילים את השאילתות שלהם באופן מיידי ומחזירים ערך, ולא אובייקט סָפיר, למטרת איטרציה בהמשך.

לדוגמה, הקוד הבא ממחיש את השימוש באופרטורים Sum ו-Count, אשר מחזירים שלמים (int). שים לב לנקודות הבאות בנוגע לקוד:

- האופרטורים מופעלים בתור שיטות, **באופן ישיר על אובייקטי הרצף**, אשר במקרה שלנו הינם מערך **numbers**.
- הסוג המוחזר אינו אובייקט IEnumerable, אלא **int**.

```
static void Main(string[] args)
{
    var myNumbers = new[] { 1,2,3 };

    int myTotalNumbers = myNumbers.Sum();
    int myNumbersQuantity = myNumbers.Count();

    Console.WriteLine("I have {0} numbers, and their sum = {1}"
                            , myNumbersQuantity, myTotalNumbers);

}
```

קוד זה יפיק את הפלט הבא:

```
I have 3 numbers, and their sum = 6
```

יש 47 אופרטורי שאילתה סטנדרטיים אשר משתייכים ל-14 קטגוריות שונות. קטגוריות אלו מוצגות בטבלה 30-1.

טבלה 30-1: קטגוריות של אופרטורי שאילתה סטנדרטיים

סוג אופרטור	אופרטור למטרת	כמות אופרטורים	תיאור
Restriction	הגבלה	1	מחזיר קבוצה של אובייקטים מהרצף על סמך קריטריון בחירה.
Projection	הקרנה	2	בוחר איזה חלקים מהאובייקטים של הרצף יאוחזרו.
Partitioning	הפרדה	4	מחזיר אובייקטים מהרצף או מדלג עליהם.
Join	צירוף	2	מחזיר אובייקט IEnumerable אשר משמש לצירוף שני רצפים על סמך קריטריון כלשהו.
Concatenation	שרשור	1	מפיק רצף יחיד משני רצפים נפרדים.
Ordering	סידור	2	מסדר את הרצף על סמך קריטריון נתון.
Grouping	קיבוץ	1	מקבץ את הרצף על סמך קריטריון נתון.
Set	קביעה	4	מבצע פעולות קביעה ברצף.
Conversion	המרה	7	ממיר רצפים למספר תצורות שונות, כגון מערכים, רשימות ומילונים.
Equality	השוואה	1	משווה שני רצפים ומוצא התאמות.
Element	אלמנט	9	מחזיר אלמנט מסוים מהרצף.
Generation	חילול	3	מחולל רצפים.
Quantifiers	מכמת	3	מחזיר ערכים בוליאניים אשר מציינים אם הרצף מקיים תנאי כלשהו.
Aggregate	התקבצות	7	מחזיר ערך יחיד אשר מייצג מאפיינים של הרצף.

סיכום

שפת השאילתות LINQ היא ללא ספק הפסגה שאליה מכוונים מירב המאפיינים החדשים של C# 3.0.

בפרק זה סקרנו את התחביר הבסיסי של LINQ, והצגנו את המאפיינים החדשים של השפה כדי להדגים את יכולות הטכנולוגיה החדשה.

סקרנו את ספקי LINQ השונים, עברנו על סוגי התחבירים, סוגי תוצאות, מבנה ביטויי שאילתה ועל הפסקאות המרכיבות אותן.

באתר של מיקרוסופט ניתן למצוא 101 דוגמאות לשימוש ב-LINQ בכתובת הבאה:

http://msdn.microsoft.com/en-us/vcsharp/aa336746.aspx

פרק 31

LINQ to SQL

LINQtoSQL הינו רכיב של net framework 3.5., שמטרתו לספק תשתית זמן ריצה לניהול נתונים טבלאיים (מבסיס הנתונים) ומאפשר התייחסות אליהם כאובייקטים.

בזמן הפיתוח מפתחים ישויות כמו לקוח, מוצר, הזמנה. ישויות אלו הן למעשה מופע של מחלקות שיש להן מאפיינים ושיטות. כאשר יש לטעון, לעדכן ולשמור ישויות אלו, זהו התפקיד של המְפַתֵח לכתוב קוד המתרגם ישויות אלו אל בסיס הנתונים. הוא עושה זאת על ידי כתיבת משפטי SQL ותרגום המאפיינים השונים של הישויות לסוג נתונים שונים בטבלאות שבבסיס הנתונים.

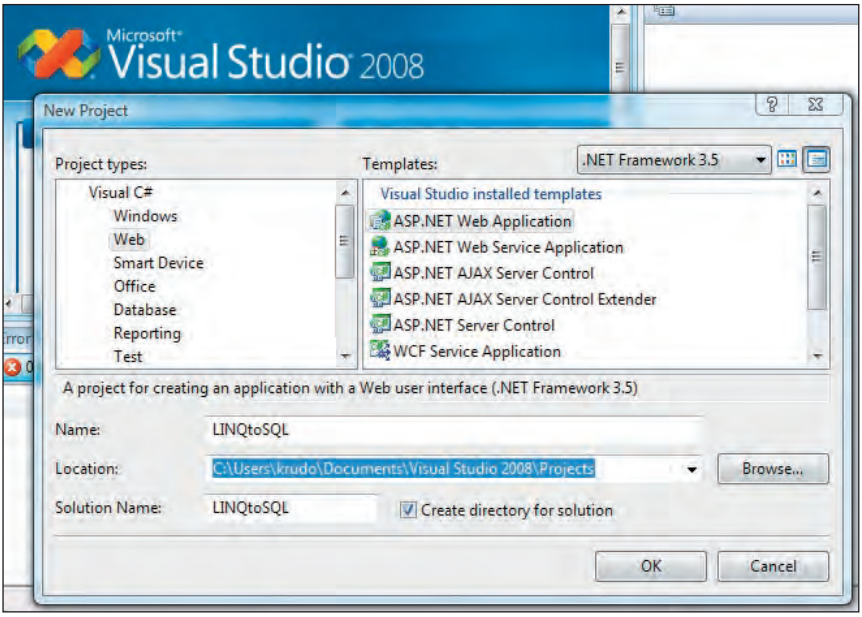
הרעיון של LINQtoSQL הוא מיפוי מודל הנתונים הטבלאי למודל אובייקטים. כאשר המערכת פועלת, היא מתרגמת את הפעולות שבוצעו על מודל האובייקטים לפקודות SQL, שולחת את השאילתות לבסיס הנתונים ומריצה אותן. כאשר בסיס הנתונים מחזיר נתונים, רכיב LINQtoSQL מתרגם שוב את הנתונים דרך המיפוי שהוגדר מבעוד מועד אל תוך מודל האובייקטים.

אנו רוצים לפעול עם נתונים ממקורות מידע שונים באותה דרך תחת אותה סביבת עבודה.

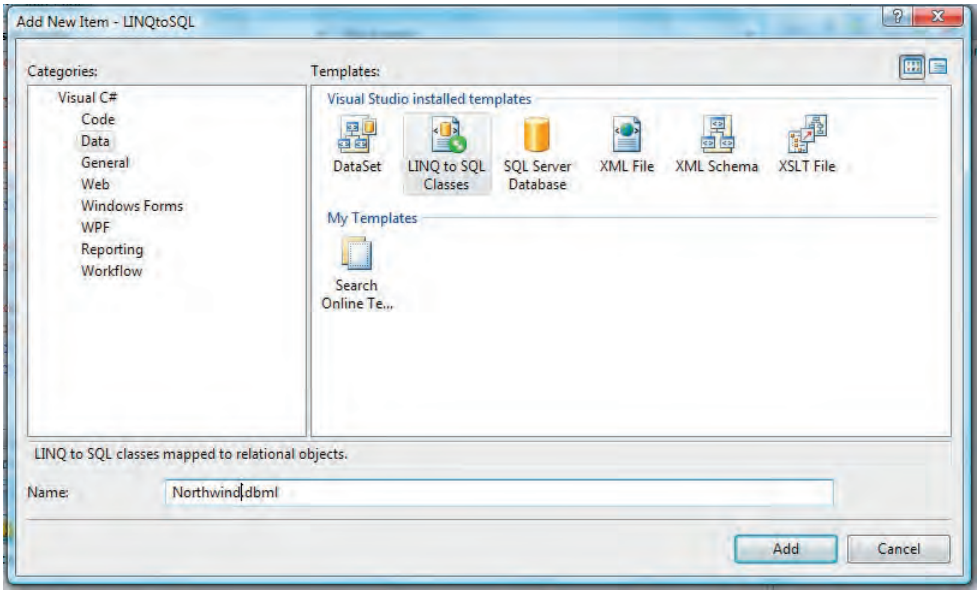
סביבת הפיתוח החדשה **Visual Studio 2008** מגיעה עם כלי עיצוב חדש התומך ב-LINQtoSQL. כלי זה מאפשר לעצב ולהציג בצורה חזותית מבנה נתונים טבלאי (**SQL Server 2005**) כמודל אובייקטים הנתמך כתצורת עבודה של LINQ to SQL.

בדוגמאות הבאות נשתמש בבסיס הנתונים **Northwind** אשר הכרנו בפרק 23.

פתח את Visual Studio 2008. כעת ניצור פרויקט חדש מסוג **ASP.NET Web Application** בשם LINQtoSQL:



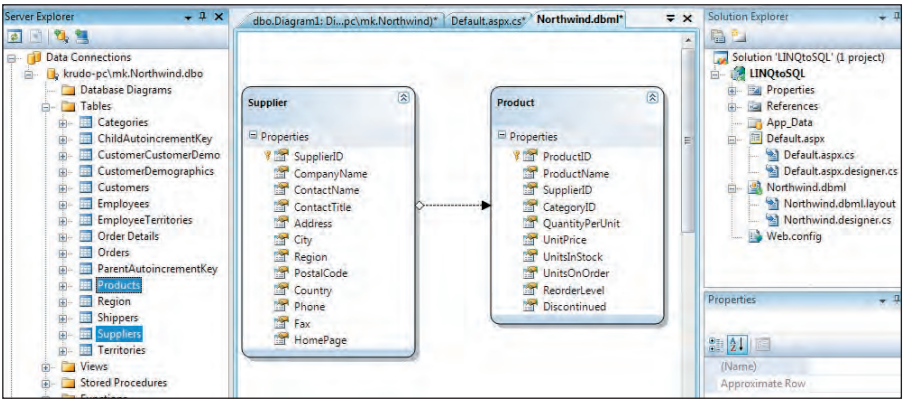
כעת נוסיף את המעצב לסביבת הפיתוח: לחץ על **Project** ← **Add new Item** או על שילוב המקשים: Ctrl+Shift+A ובחר **להוסיף תבנית LINQ to SQL Classes** תחת קטגוריה **Data**. בתחתית המסך, שנה את השם ל- **Northwind.dbml**.



בסייר Windows לחץ על הקובץ **Northwind.dbml** ופתח אותו (אם אינו פתוח כבר). זהו המעצב שיש לו משטח עבודה, ונוסיף לו את הטבלאות הנחוצות אשר נחקור ונעדכן.

פתח את Server Explorer בלחיצה על **View** ← **Server Explorer**, הוסף מתוכו קישור אל בסיס הנתונים Northwind, כפי שלמדנו בפרק 23.

לאחר הקישור לבסיס הנתונים, גרור אל משטח העבודה את הטבלאות: **Products** ו-**Suppliers**.



המעצב מכיל שני אובייקטים במודל הנתונים, שמייצגים מופע בודד של הטבלאות שגררנו אל משטח העבודה. שים לב לשמות שלהם: Product ו-Supplier. סביבת העבודה של Visual Studio 2008 יודעת, באופן מוחלט כמעט, להפוך שמות של טבלאות מרבים ליחיד (לפי חוקי הדקדוק של השפה האנגלית) כמו Products ← Product. ניתן כמובן לשנות את השמות על ידי לחיצה על הכותרת ועריכת שינוי השם. כמו כן ניתן לשנות שמות או להסיר מאפיינים (עמודות) מהאובייקט אם אין לנו צורך בהם.

בנוסף, המעצב מציג גם את הקישור בין האובייקטים, ממש כפי שיושם בבסיס הנתונים. לחיצה על הקשר בין האובייקטים ולחיצה נוספת על המקש **F4** תראה לנו את מאפייני הקשר. בדוגמה שלנו הקשר הוא יחיד לרבים, בין Products לבין Supplier, שבו עבור Supplier אחד יש אחד או כמה Products.

בעת לחיצה על **Save** לאחר התאמת האובייקטים, בחן את הקובץ **Northwind.designer.cs**. קובץ קוד זה מכיל מחלקות המייצגות את מה שנבנה על משטח העבודה, שים לב למאפיינים או התכונות (Attributes) שבתוך הקובץ, אשר משקפים את היחס במיפוי בין האובייקטים שנוצרו בקוד לבין הישויות בבסיס הנתונים. אנו נשתמש במחלקות אלו מייד, כאשר נעבוד עם LINQ.

שליפת נתונים באמצעות מודל הנתונים

נפתח את הקובץ **Default.aspx** ונוסיף לו ארבעה לחצנים ואובייקט **GridView** אחד. נעשה זאת על ידי הקוד הבא:

```
<asp:Button ID="Select" runat="server" Text="Select" />
<asp:Button ID="Insert" runat="server" Text="Insert" />
<asp:Button ID="Update" runat="server" Text="Update" />
<asp:Button ID="Delete" runat="server" Text="Delete" />
<br />
<hr />
<asp:GridView ID="grdProducts" runat="server">
</asp:GridView>
```

נעבור לצד השרת, ונוסיף את השיטה הבאה:

```
private void ShowProducts()
{
    // יצירת מופע של אוביקט מודל הנתונים
    NorthwindDataContext db = new NorthwindDataContext();

    // שימוש תקני ב-LINQ כאשר מקור הנתונים הינו מודל הנתונים DB
    var products = from p in db.Products
                   where p.Supplier.CompanyName == "Tokyo Traders"
                   orderby p.ProductID descending
                   select new {p.ProductID, p.ProductName,
                    p.Supplier.CompanyName, Price = p.UnitPrice };

    // חיבור הנתונים לרכיב הטבלה
    grdProducts.DataSource = products;
    // הצגת הנתונים
    grdProducts.DataBind();
}
```

כפי שניתן לראות, נעשה כאן שימוש תקני ב-LINQ. עם זאת, באופן שונה מהפרק הקודם, מקור הנתונים מגיע ממודל הנתונים שיצרנו בעזרת המעצב.

השאילתה מבקשת לקבל נתונים מאובייקט Products (אשר כזכור ממופה לטבלה Products בבסיס הנתונים). השדה **CompanyName** המשמש לקישור בין האובייקטים הוא "Tokyo Traders". בנוסף, נבקש שהנתונים ימוינו בסדר יורד לפי מספר המוצר (ProductID).

לחיצה כפולה על לחצן **select** במסך תיצור את אירוע הלחיצה. נוסיף לאירוע את הקריאה לשיטה **ShowProducts**. אחר כך נפעיל את האתר על ידי הקשה על **F5** (באם יופיע חלון עם כותרת: "Script Debugging Disabled" לחץ yes). לאחר לחיצה על הלחצן select, תוצג רשימת מוצרים אשר הספק שלהם הוא Tokyo Traders:

הוספת נתונים באמצעות מודל הנתונים

כעת נוסיף את הקוד הבא לאירוע של הלחצן Insert:

```
protected void Insert_Click(object sender, EventArgs e)
{
    // יצירת מופע של אובייקט מודל הנתונים
    NorthwindDataContext db = new NorthwindDataContext();

    // יצירת אובייקט חדש (שימוש באתחול אובייקט)
    Product newProduct = new Product { SupplierID = 4,
                                       ProductName = "Nagashi",
                                       UnitPrice = 2.90m };
    // הכנסת האובייקט החדש למודל הנתונים
    db.Products.InsertOnSubmit(newProduct);

    // עדכן נתונים
    db.SubmitChanges();
    // הצגת הנתונים
    ShowProducts();
}
```

לאחר יצירת מופע של מודל הנתונים, צריך ליצור אובייקט חדש ולהשתמש ב'סוג אנונימי' וב'אתחול האובייקט'. נכניס את האובייקט החדש לאוסף המוצרים של מודל הנתונים על ידי שימוש בשיטה **InsertOnSubmit**. לאחר הקריאה לשיטה **SubmitChanges** הנתונים יישמרו בבסיס הנתונים. כעת נפעיל את האתר על ידי הקשה על **F5**. נסיים בלחיצה על **Insert**.

לפנינו מוצגת רשימת המוצרים אשר כוללת את המוצר החדש שזה עתה הוספנו:

Select Insert Update Delete

ProductID	ProductName	CompanyName	Price
78	Nagashi	Tokyo Traders	2.9000
74	Longlife Tofu	Tokyo Traders	10.0000
10	Ikura	Tokyo Traders	31.0000
9	Mishi Kobe Niku	Tokyo Traders	97.0000

עדכון נתונים באמצעות מודל הנתונים

הוסף את הקוד הבא לאירוע של הכפתור **Update**:

```
protected void Update_Click(object sender, EventArgs e)
{
    // יצירת מופע של אובייקט מודל הנתונים
    NorthwindDataContext db = new NorthwindDataContext();

    // איתור המוצר לעדכון (שימוש בביטוי למבדא)
    Product product = db.Products.First(p =>
                      p.ProductName.Equals("Nagashi"));
    // עדכן מחיר מוצר
    product.UnitPrice += 0.10m;
    // עדכן נתונים
    db.SubmitChanges();
    // הצגת הנתונים
    ShowProducts();
}
```

לאחר יצירת מודל הנתונים, צריך לאתר את המוצר שיש לעדכן. עושים זאת על ידי שימוש בביטוי למבדא. בשורה שלאחר מכן מעדכנים את מחיר המוצר ובסופו של דבר מעדכנים גם את הנתונים בבסיס הנתונים. נפעיל את האתר על ידי הקשה על **F5** בחלון שיופיע. נסיים בלחיצה על **Update**.

לפנינו מוצגת רשימת המוצרים אשר כוללת את המוצר החדש שעודכן זה עתה:

Select Insert Update Delete

ProductID	ProductName	CompanyName	Price
78	Nagashi	Tokyo Traders	3.0000
74	Longlife Tofu	Tokyo Traders	10.0000
10	Ikura	Tokyo Traders	31.0000
9	Mishi Kobe Niku	Tokyo Traders	97.0000

מחיקת נתונים באמצעות מודל הנתונים

הוסף את הקוד הבא לאירוע של הכפתור **Delete**:

```
protected void Delete_Click(object sender, EventArgs e)
{
    // יצירת מופע של אובייקט מודל הנתונים
    NorthwindDataContext db = new NorthwindDataContext();

    // איתור המוצר לעדכון (שימוש בביטוי למבדא)
    Product product = db.Products.First(p =>
                      p.ProductName.Equals("Nagashi"));
    // מחק את המוצר ממודל הנתונים
    db.Products.DeleteOnSubmit(product);
    // עדכן נתונים
    db.SubmitChanges();
    // הצגת הנתונים
    ShowProducts();
}
```

לאחר יצירת מודל הנתונים, נאתר את המוצר שיש למחוק, בשורה שלאחר מכן נמחק אותו ממודל הנתונים. בסופו של דבר נעדכן את הנתונים בבסיס הנתונים. נפעיל את האתר על ידי הקשה על **F5**. נסיים בלחיצה על **Delete** כדי למחוק את המוצר.

לפנינו מוצגת רשימת המוצרים לאחר מחיקת המוצר:

Select Insert Update Delete

ProductID	ProductName	CompanyName	Price
74	Longlife Tofu	Tokyo Traders	10.0000
10	Ikura	Tokyo Traders	31.0000
9	Mishi Kobe Niku	Tokyo Traders	97.0000

ניפוי שאילתה במודל הנתונים

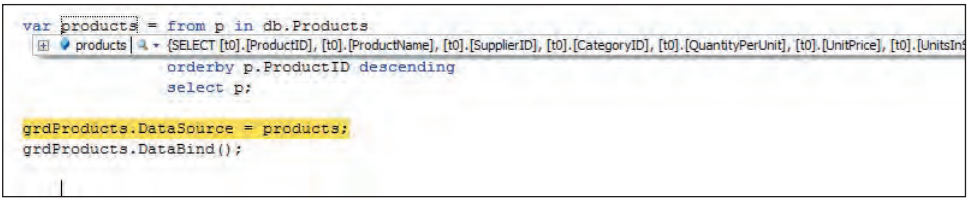
בדוגמה הבאה נשתמש בכלי העזר **LINQ to SQL Debug Visualizer** אשר מאפשר לראות נתוני משתנה שמחזיק את שאילתת SQL שתופעל על בסיס הנתונים, ותחזיר את הנתונים שביקשנו. ניתן לקרוא ולהוריד את כלי העזר חינם מאתר הבלוג של סקוט בכתובת:

http://weblogs.asp.net/scottgu/archive/2007/07/31/linq-to-sql-debug-visualizer.aspx

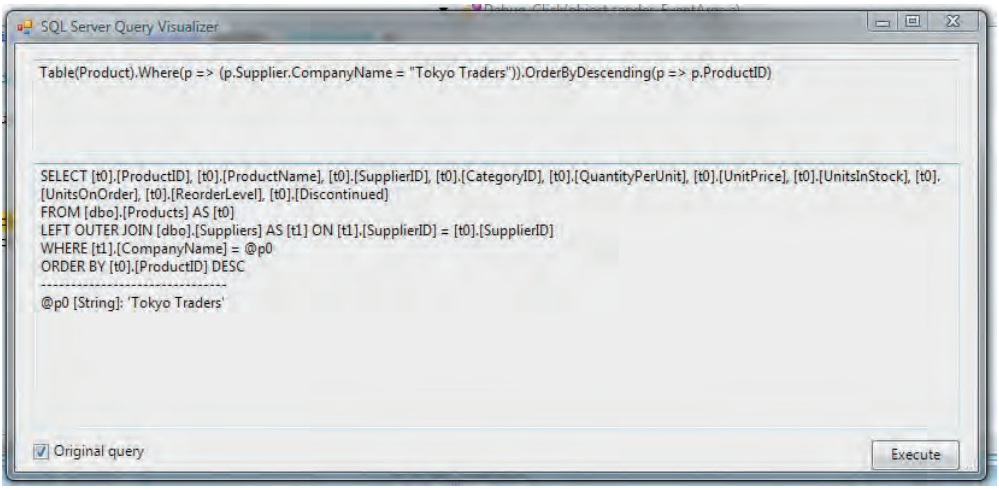
הבא נבחן מה מתרחש מאחורי הקלעים. כאשר אנו מבקשים נתונים ממודל הנתונים, חייבת להיות ללא ספק דרך כלשהי שבה משתמש המעצב להביא את הנתונים מבסיס הנתונים. נוסיף לחצן על המסך ונקבע את שמו ל-"Debug". לאירוע הלחיצה שלו בצד השרת צריך להוסיף את הקוד הבא:

```
protected void Debug_Click(object sender, EventArgs e)
{
    var products = from p in db.Products
                   where p.Supplier.CompanyName == "Tokyo Traders"
                   orderby p.ProductID descending
                   select p;

    grdProducts.DataSource = products;
    grdProducts.DataBind();
}
```

מקם את סמן העכבר על השורה grdProducts.DataSource = products;, והצב נקודת עצירה על ידי הקשה על המקש **F9**. הרץ שוב את האתר על ידי הקשה על **F5**, לחץ על כפתור **Update**. המערכת נעצרה בנקודת העצירה בשורה שסומנה. הבא נבחן את המשתנה products על ידי שימוש בכלי העזר **LINQ to SQL Debug Visualizer** לקריאת נתוני משתנה:

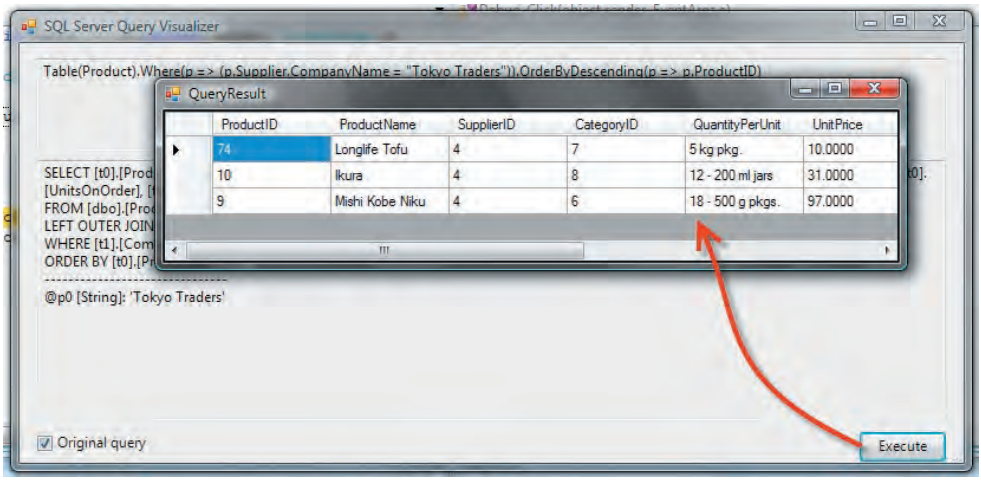


לחץ על זכוכית המגדלת כדי לראות את נתוני המשתנה בפורמט קריא יותר:



כעת ניתן לראות בבירור את השאילתה שתבוצע על בסיס הנתונים. ניתן לראות אף את הקשר הקיים בין הטבלאות Products ו-Supplier.

לחיצה על תיבת הסימון **Original Query** אף תראה את הפרמטר שמועבר לשאילתה. ניתן להפעיל את השאילתה על ידי לחיצה על הלחצן **Execute** ולקבל את תוצאות השאילתה מוצגים בטבלה:



סיכום

בפרק זה הוצג רכיב התוכנה **LINQ to SQL**, אשר באמצעותו ניתן לפתח מערכת מידע בסביבה אחת תוך שימוש ב- Visual Studio 2008. סביבת פיתוח זו כוללת כלי עיצוב חדש התומך ב- LINQ to SQL. כפי שראינו, כלי עיצוב זה מאפשר **למפות מודל נתונים טבלאי אל מודל אובייקטים**. כך, רוב העבודה על בסיס הנתונים נעשית באמצעות מודל האובייקטים.

פרק 32

LINQ to XML

לאחרונה הפכה השפה XML (Extensible Markup Language) לאמצעי חשוב לאחסון והחלפה של נתונים. השפה C# 3.0 כוללת מאפיינים חדשים אשר מקלים רבות על העבודה עם XML בהשוואה לשיטות ישנות יותר, כגון XPath ו-XSLT. אם שיטות אלו מוכרות לך, בוודאי תשמח לשמוע שהאפשרות LINQ to XML מפשטת את העבודה עם XML במספר דרכים שונות, וביניהן:

- ניתן ליצור עץ XML מלמעלה למטה במשפט יחיד.
- ניתן ליצור ולתפעל XML בזיכרון ללא צורך במסמך XML לאחסון העץ.
- ניתן ליצור ולתפעל רכיבי מחרוזת ללא צורך ברכיב מחרוזת משני (מסוג ענף).

להלן הקדמה קצרה בנושא, לפני שנעבור לתיאור חלק מאפשרויות תפעול XML החדשות של C# 3.0.

שפות סימני עריכה

XML, שפת סימני עריכה, הינה סדרה של תוויות אשר נמצאות במסמך כדי לספק מידע על המידע שנמצא במסמך. כלומר, תוויות סימני העריכה אינן נתוני המסמך עצמם, אלא מכילות נתונים על הנתונים. **נתונים על נתונים** נקראים **metadata**.

שפת סימני עריכה היא סדרה מוגדרת של תוויות שתפקידן להעביר סוגים מסוימים של metadata על תוכן המסמך. **HTML**, לדוגמה, היא שפת סימני העריכה המוכרת ביותר. התוויות של metadata מכילות מידע על אופן הצגת דף האינטרנט על ידי הדפדפן, ואופן הניווט בין הדפים השונים באמצעות קישורי היפרטקסט.

בשעה שרוב שפות סימני העריכה כוללות סדרת תוויות מוגדרת מראש, XML כוללת מספר קטן של תוויות מוגדרות, ואילו שאר התוויות מוגדרות על ידי המתכנת על פי סוגי ה-metadata שעליהן לייצג בסוג המסמך הרצוי. כל עוד יש הסכמה בין הכותבים והקוראים של הנתונים בנוגע למשמעותן של התוויות, אין שום בעיה להוסיף לאותן תוויות כל מידע שימושי שהמעצב צריך.

עקרונות XML

הנתונים במסמכי XML מאוחסנים בעץ XML, אשר בעיקרו מורכב מסדרה של אלמנטים מקוננים.

אלמנט הינו המרכיב הבסיסי של **עצי XML**. לכל אלמנט יש שם, והוא יכול להכיל נתונים. חלק מהם יכולים להכיל גם אלמנטים מקוננים אחרים. האלמנטים מתוחמים באמצעות תוויות פותחות וסוגרות. כל הנתונים אשר מאוחסנים באלמנט חייבים להימצא בין התווית הפותחת לבין התווית הסוגרת.

- **תווית פותחת** מתחילה בסימן '<', אחריו שם האלמנט, אחריו (אופציונלי) מספר בלתי מוגבל של מאפיינים (attributes) ולבסוף הסימן '>'.

```
<Employee>
```

- **תווית סוגרת** מתחילה עם סימון '</' , אחריו שם האלמנט ולבסוף הסימן '>'.

```
</Employee>
```

- **אלמנט חסר תוכן** יוצג כך:

```
<Employee />
```

במקטע XML שלהלן מוצג אלמנט בשם **ProductID**, ואחריו אלמנט בשם **ProductName**, ואחריו אלמנט ריק בשם **ProductPrice**.

```
<ProductID>98765</ProductID>
<ProductName>ThinkPad X61<ProductName>
<ProductPrice />
```

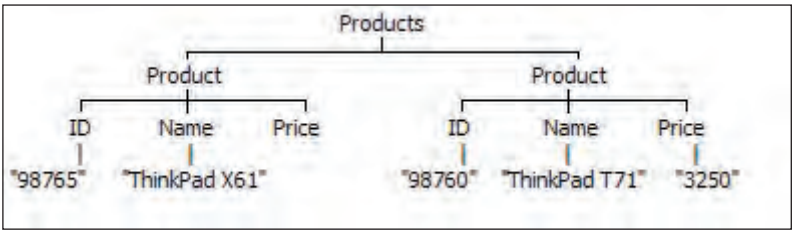
להלן מספר נקודות חשובות נוספות שעליך לדעת על XML:

- **מסמכי XML** חייבים לכלול אלמנט ראשי יחיד, אשר מכיל את כל האלמנטים האחרים.
- **תוויות XML** חייבות להיות מקוננות כהלכה.
- בניגוד לתוויות HTML, תוויות XML רגישות לרישיות (caps).
- מאפייני XML (attributes) הינם שם/ערך שמכילים **metadata** נוספים על האלמנט. חובה לתחום את חלק הערך של המאפיין משני צדיו על ידי גרשיים או על ידי גרש אחד.
- במסמכי XML הרווחים נשמרים, בניגוד למסמכי HTML, שבהם הרווחים מאוחדים לרווח יחיד בפלט.

מסמך XML שלהן הינו דוגמה לקוד XML אשר מכיל מידע על שני מוצרים. עץ XML זה פשוט ביותר, כדי לאפשר הצגה ברורה של האלמנטים. הנה מספר נקודות חשובות שעליך לדעת על עץ XML זה:

- העץ כולל רכיב שורש מסוג Products, שמכיל שני רכיבי משנה מסוג Product.
- כל רכיב Product מכיל רכיבים, שמכילים את השם ומספר המוצר ואת מחירו.

```
<Products>
   <Product>
      <ID>98765</ID>
      <Name>ThinkPad X61</Name>
      <Price />
   </Product>
   <Product>
      <ID>98760</ID>
      <Name>ThinkPad T71</Name>
      <Price>3250</Price>
   </Product>
</Products>
```



תרשים 32-1: המבנה ההיררכי של העץ XML שבדוגמה

מחלקות XML

ניתן להשתמש ב-**LINQ** ל-**XML** עבור קוד XML בשתי דרכים שונות. הדרך הראשונה היא גרסה מפושטת של הממשק לתפעול יישומים (**XML LINQ API**). הדרך השנייה היא שימוש בכלי השאילתה של LINQ.

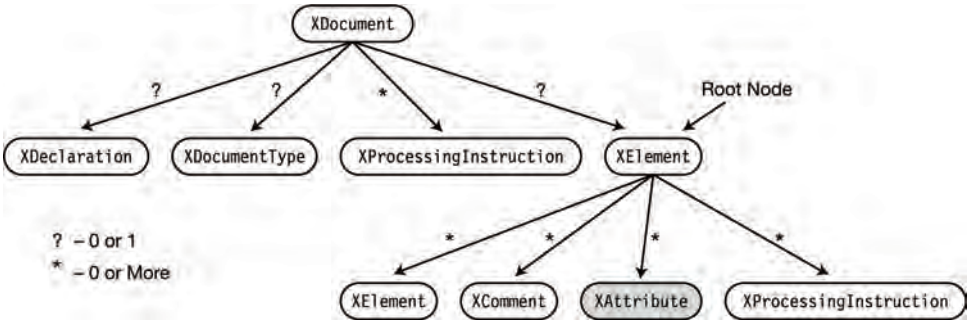
הממשק לתכנות יישומים של LINQ ל-XML מורכב ממספר מחלקות אשר מייצגות את המרכיבים של עץ XML. שלוש המחלקות החשובות ביותר שתפעיל יהיו **XElement**, **XAttribute** ו-**XDocument**. יש מחלקות נוספות, אולם אלו הן העיקריות.

בתרשים 32-1 ניתן לראות שעץ XML הוא סדרה של אלמנטים מקוננים. בתרשים 32-2 מוצגות המחלקות המשמשות לבניית עץ XML.

בתרשים מוצגות הנקודות הבאות:

- רכיבי המשנה הישירים האפשריים עבור **רכיב XDocument**:
  - עד אחד מסוגי הרכיבים הבאים: רכיב **XDeclaration**, רכיב **XDocumentType** ורכיב **XElement**.
  - מספר בלתי מוגבל של רכיבי **XProcessingInstruction**.
- כאשר ישנו רכיב **XElement** עליון תחת **XDocument**, הוא יתפקד כשורש של כל שאר המרכיבים של עץ XML.

- אלמנט השורש יכול להכיל מספר בלתי מוגבל של רכיבי **XElement**, **XComment** או **XprocessingInstruction**, בתוך כל אחד מהשלבים.



תרשים 32-2: מבנה המחלקות של רכיבי XML

למעט המחלקה XAttribute, רוב המחלקות המשמשות ליצירת עץ XML נגזרות ממחלקה בשם XNode, ובספרות המקצועית הן נקראות באופן כוללני "XNode". בתרשים 32-2, מחלקות XNodes מוצגות במסגרות אפורות, והמחלקה XAttribute מוצגת במסגרת אפורה.

יצירה, שמירה, טעינה והצגה של מסמכי XML

הדרך הטובה ביותר להדגים את אופן השימוש ואת הפשטות של **XML LINQ API** הינה על ידי הצגת דוגמאות קוד פשוטות. לדוגמה, הקוד שלהלן ממחיש כמה פשוט לבצע חלק מהמטלות החשובות הנדרשות בעת עבודה עם XML.

הקוד מתחיל ביצירת עץ XML פשוט אשר כולל את הרכיב Products, עם שני רכיבי משנה שמכילים את שמות שני המוצרים. שים לב למספר נקודות חשובות בקוד:

- העץ נוצר באמצעות משפט יחיד, אשר בתורו יוצר את כל האלמנטים המקוננים אשר מרכיבים את העץ. תהליך זה מכונה **בנייה פונקציונלית**.
- כל אלמנט נוצר במיקומו באמצעות ביטוי יצירת-אובייקטים, ובאמצעות הבנאי של סוג הרכיב.

לאחר יצירת העץ, הקוד שומר אותו לקובץ ProductsFile.xml על ידי השיטה **Save** של XDocument. לאחר מכן הוא טוען שוב את עץ XML מהקובץ על ידי השיטה הסטטית **Load** של XDocument, ומקצה את העץ לאובייקט XDocument חדש. בסופו של דבר הוא משתמש בשיטה **WriteLine** להצגת מבנה העץ שמאוחסן באובייקט XDocument החדש.

```
class Program
{
 static void Main()
 {
  XDocument products =
            new XDocument(                          // יצירת מסמך
             new XElement("Products",               // יצירת אלמנט ראשי
              new XElement("Name", "ThinkPad X61"), // יצירת אלמנט
              new XElement("Name", "ThinkPad T71")  // יצירת אלמנט
              )
            );
  products.Save("ProductsFile.xml");                // שמירה לקובץ

  XDocument LoadedProducts = XDocument.Load("ProductsFile.xml");

  Console.WriteLine(LoadedProducts);                // הצגת המסמך
 }
}
```

קוד זה יפיק את הפלט הבא:

```
<Products>
   <Name>ThinkPad X61</Name>
   <Name>ThinkPad T71</Name>
</Products>
```

יצירת עץ XML

בדוגמה הקודמת למדת שבאפשרותך ליצור מסמך XML בזיכרון על ידי שימוש בבנאים עבור **XDocument** ו-**XElement**. בשני סוגי הבנאים נכתוב כך את שני הפרמטרים:

- הפרמטר הראשון הוא שם האובייקט.
- הפרמטר השני ואילך מכילים את רכיבי עץ XML. הפרמטר השני של הבנאי הוא הפרמטר **params**, ולכן יכול לקבל מספר בלתי מוגבל של פרמטרים.

לדוגמה, הקוד שלהלן יוצר עץ XML ומציג אותו באמצעות השיטה **Console.WriteLine**:

```
class Program
{
  static void Main()
  {
    XDocument products =
              new XDocument(
                new XElement("Products",

                   new XElement("Product",
                       new XElement("ID","98765"),
                       new XElement("Name" ,"ThinkPad X61")),
```

```
                    new XElement("Product",
                        new XElement("ID","98760"),
                        new XElement("Name" ,"ThinkPad X71"),
                        new XElement("Price","3250"))

                    )
                );

    Console.WriteLine(products);
  }
}
```

קוד זה יפיק את הפלט הבא:

```
<Products>
  <Product>
    <ID>98765</ID>
    <Name>ThinkPad X61</Name>
  <Product/>
  <Product>
    <ID>98760</ID>
    <Name>ThinkPad X71</Name>
    <Price>3250</Price>
  <Product/>
</Products>
```

שימוש בערכים מעץ XML

היתרון העיקרי של XML מורגש במיוחד כאשר עליך לסרוק את עץ XML ולשלוף או לשנות ערכים. השיטות העיקריות המשמשות לשליפת נתונים מוצגות בטבלה 32-1.

טבלה 32-1: שיטות חקירת XML

שם השיטה	מחלקה	סוג מוחזר	תיאור
Nodes	XDocument XElement	IEnumerable<object>	מחזירה את כל הבנים של הרכיב הנוכחי, ללא התניית סוג.
Elements	XDocument XElement	IEnumerable<XElement>	מחזירה את כל הבנים XElement של הרכיב הנוכחי, או את כל הבנים בעלי שם מסוים.
Element	XDocument XElement	XElement	מחזירה את הבן/ילד XElement הראשון של הרכיב הנוכחי, או את הבן הראשון עם שם מסוים.

שם השיטה	מחלקה	סוג מוחזר	תיאור
Descendents	XElement	IEnumerable<XElement>	מחזירה את כל צאצאי XElement של הרכיב הנוכחי, או את כל צאצאי XElement עם שם מסוים, ללא תלות במספר הרמות שהם נמצאים תחת הרכיב הנוכחי.
DescendentsAndSelf	XElement	IEnumerable<XElement>	זהה ל-Descendents, ובנוסף לצאצאים מחזירה גם את הרכיב הנוכחי.
Ancestors	XElement	IEnumerable<XElement>	מחזירה את כל האבות XElement, או את כל האבות XElement שמעל הרכיב הנוכחי ובעלי שם מסוים.
AncestorsAndSelf	XElement	IEnumerable<XElement>	זהה ל-Ancestors, ובנוסף לאבות מחזירה גם את הרכיב הנוכחי.
Parent	XElement	XElement	מחזיר את ההורה של הרכיב הנוכחי.

להלן מספר נקודות חשובות שעליך לדעת על השיטות שמוצגות בטבלה 1-32:

- **Nodes**: השיטה Nodes מחזירה אובייקט מסוג IEnumerable<object>, מכיוון שסוג הרכיב המוחזר אינו קבוע ויכול להיות XElement, XComment וכד'. תוכל להשתמש בשיטה OfType<Type> כדי לקבוע את סוג הרכיבים שיוחזרו. לדוגמה, שורת הקוד הבאה שולפת רכיבי XComment בלבד:

```
IEnumerable<XComment> comments = xd.Nodes().OfType<XComment>();
```

- **Elements**: מכיוון ששליפת XElement היא תהליך שכיח למדי, יש קיצור דרך לביטוי Nodes().OfType<XElement>() – השיטה Elements.
  - שימוש בשיטה Elements ללא פרמטרים יגרום להחזרת כל הבנים XElement.
  - שימוש בשיטה Elements עם פרמטר שם יחיד יגרום להחזרת הבנים XElement עם שם זה בלבד. לדוגמה, שורת הקוד שלהלן תחזיר את כל הבנים XElement בשם Price.

```
IEnumerablr<XElement> prodPrice = prod.Elements("Price");
```

- **Element**: שיטה זו שולפת רק את הבן XElement הראשון של הרכיב הנוכחי. בדומה לשיטה Elements, ניתן לבצע את הקריאה לשיטה עם פרמטר יחיד או ללא פרמטרים

כלל. ללא פרמטרים, השיטה תחזיר את הבן XElement הראשון. עם פרמטר יחיד, היא תחזיר את הבן XElement הראשון בשם המצוין.

- **Descendents** ו-**Ancestors**: שיטות אלו פועלות באופן דומה לשיטות Elements ו-Parent, אולם במקום להחזיר את הבנים או ההורים הישירים, הן כוללות גם אלמנטים שנמצאים מתחת או מעל הרכיב הנוכחי, ללא תלות בהבדל ברמות הקינון.

הקוד הבא ממחיש את אופן השימוש בשיטות **Element** ו-**Elements**:

```
static void Main()
{
  XDocument products =
            new XDocument(
              new XElement("Products",

                new XElement("Product",
                    new XElement("ID","98765"),
                    new XElement("Name" ,"ThinkPad X61"),
                    new XElement("Contact" ,"David"),
                    new XElement("Contact" ,"Meir")),


                new XElement("Product",
                    new XElement("ID","98760"),
                    new XElement("Name" ,"ThinkPad X71"),
                    new XElement("Price","3250"),
                    new XElement("Contact","Ravid"))

                )
            );

  // Products קבל את האלמנט הראשון בשם
  XElement root = products.Element("Products");
  IEnumerable<XElement> myProducts = root.Elements();

  foreach (XElement myProd in myProducts)
  {
     // Name קבל את האלמנט הראשון בשם
     XElement prodNameNode = myProd.Element("Name");
     Console.WriteLine(prodNameNode.Value);

     // Contact קבל את האלמנט הראשון בשם
     IEnumerable<XElement> prodContact = myProd.Elements("Contact");
     foreach (XElement  contact in prodContact )
     {
        Console.WriteLine("   {0}", contact.Value);
     }
  }

}
```

קוד זה יפיק את הפלט הבא:

```
ThinkPad X61
   David
   Meir
ThinkPad X71
   Ravid
```

הוספת רכיבים ותפעול XML

השיטה **Add** מאפשרת להוסיף בנים לאלמנט קיים. היא גם מאפשרת להוסיף כמה אלמנטים בקריאה אחת לשיטה, ללא הגבלה על סוגי הרכיבים המוספים.

לדוגמה, הקוד שלהלן יוצר עץ XML פשוט ומציג אותו. לאחר מכן הוא משתמש בשיטה Add כדי להוסיף רכיב בודד לאלמנט השורש. אחר כך הוא משתמש בשיטה Add פעם נוספת כדי להוסיף שלושה אלמנטים - שניים מסוג **XElement** ואחד מסוג **XComment**. שים לב לתוצאות שמתקבלות בפלט:

```
static void Main(string[] args)
{
   // בניית עץ
   XDocument xDoc  = new XDocument(
      new XElement("ROOT", new XElement("NODE_01")
      )
   );

   Console.WriteLine("I have one node:");
   Console.WriteLine(xDoc );

   // קבל הפניה לעץ
   XElement myRoot = xDoc .Element("ROOT");

   // הוספת בן בודד לעץ
   myRoot.Add( new XElement("NODE_02"));

   // הוספת מספר בנים לעץ
   myRoot.Add( new XElement("NODE_03"),
               new XComment("this is a comment"),
               new XElement("NODE_04"));

   Console.WriteLine();
   Console.WriteLine(".. and now I have four nodes:");
   Console.WriteLine(xDoc );
}
```

קוד זה יפיק את הפלט הבא:

```
I have one node:
<ROOT>
   <NODE_01 />
</ROOT>

.. and now I have four nodes:
<ROOT>
   <NODE_01 />
   <NODE_02 />
   <NODE_03 />
   <!—This is a comment  --/>
   <NODE_04 />
</ROOT>
```

השיטה Add ממקמת רכיבי בנים חדשים אחרי רכיבי הבנים הקיימים, אולם תוכל להציב את הרכיבים גם לפני או בין רכיבי הבנים, על ידי השיטות **AddFirst**, **AddBeforeSelf** ו-**AddAfterSelf**.

בטבלה 2-31 מופיעות חלק מהשיטות החשובות ביותר לתפעול XML. שים לב שחלק מהן מיושמות על רכיב ההורה וחלקן על הרכיב עצמו.

טבלה 2-32: שיטות לתפעול XML

שם השיטה	קריאה מ-	תיאור
Add	רכיב אב	מוסיפה רכיב בן חדש אחרי רכיבי הבנים הקיימים של הרכיב הנוכחי.
AddFirst	רכיב אב	מוסיפה רכיב בן חדש לפני רכיבי הבנים הקיימים של הרכיב הנוכחי.
AddBeforeSelf	רכיב נוכחי	מוסיפה רכיבים חדשים לפני הרכיב הנוכחי באותה רמה.
AddAfterSelf	רכיב נוכחי	מוסיפה רכיבים חדשים אחרי הרכיב הנוכחי באותה רמה.
Remove	רכיב נוכחי	מוחקת את הרכיב הנבחר ואת התוכן שלו.
RemoveNodes	רכיב נוכחי	מוחקת את XElement הנבחר ואת התוכן שלו.
SetElment	רכיב אב	קובעת את תוכן הרכיב.
ReplaceContent	רכיב נוכחי	מחליפה את תוכן הרכיב.

עבודה עם מאפייני XML

מאפיינים מספקים מידע נוסף על רכיבי XElement, ומופיעים בתווית הפותחת של אלמנט XML.

בעת בנייה פונקציונלית של עץ XML, תוכל להוסיף מאפיינים בקלות רבה על ידי הוספת בנאיי XAttribute בתחום ההכרזה של הבנאי XElement. בנאיי XAttribute יכולים להופיע בשתי תצורות: אחת מקבלת שם וערך, והשנייה מפנה לרכיב XAttribute קיים.

הקוד שלהלן מחבר שני מאפיינים ל-**root**. שים לב ששני הפרמטרים אשר מועברים לבנאי XAttribute הם מחרוזות; הראשון מציין את שם המאפיין, והשני מעביר ערך.

```
XDocument xDoc  = new XDocument(
        new XElement("ROOT",
              new XAttribute("YEAR", "2008"), // בנאי המאפיין
              new XAttribute("TOTAL","2"),    // בנאי המאפיין
            new XElement("NODE_01"),
            new XElement("NODE_02")
        )
    );

Console.WriteLine(xDoc );
```

קוד זה יפיק את הפלט הבא. שים לב שהמאפיינים כלולים בתווית הפותחת של האלמנט:

```
<ROOT YEAR="2008" TOTAL="2">
   <NODE_01 />
   <NODE_02 />
</ROOT>
```

כדי לשלוף מאפיין מתוך רכיב XElement, עליך להשתמש בשיטה **Attribute** ולהעביר לה את שם המאפיין כפרמטר. הקוד שלהלן יוצר עץ XML עם רכיב הנושא שני מאפיינים: **TOTAL** ו-**YEAR**. לאחר מכן הוא שולף את ערכי המאפיינים ומציג אותם.

```
static void Main(string[] args)
{
    XDocument xDoc  = new XDocument(
        new XElement("ROOT",
              new XAttribute("YEAR", "2008"),
              new XAttribute("TOTAL","2"),
            new XElement("NODE_01"),
            new XElement("NODE_02")
        )
    );

    Console.WriteLine(xDoc );

    XElement myRoot = xDoc .Element("ROOT");

    XAttribute total = myRoot.Attribute("TOTAL");
    XAttribute year = myRoot.Attribute("YEAR");

    Console.WriteLine("Year  is: {0}", year.Value);
    Console.WriteLine("Total is: {0}", total.Value);
}
```

קוד זה יפיק את הפלט הבא:

```
<ROOT YEAR="2008" TOTAL="2">
   <NODE_01 />
   <NODE_02 />
</ROOT>

Year  is: 2008
Total is: 2
```

כדי להסיר מאפיין, עליך לבחור בו ולהפעיל את השיטה **Remove**, או להפעיל את השיטה **SetAttributeValue** על ההורה שלו, ואחר כך לשנות את ערך המאפיין ל-**null**. הקוד הבא מדגים את השימוש בשתי השיטות:

```
static void Main(string[] args)
{
    XDocument xDoc  = new XDocument(
        new XElement("ROOT",
            new XAttribute("YEAR", "2008"),
            new XAttribute("TOTAL","2"),
            new XElement("NODE_01"),
            new XElement("NODE_02")
        )
    );

    XElement myRoot = xDoc .Element("ROOT");

    myRoot.Attribute("TOTAL").Remove(); // הסר מאפיין TOTAL
    myRoot.SetAttributeValue("YEAR", null);    // הסר מאפיין YEAR

    Console.WriteLine(xDoc );
}
```

קוד זה יפיק את הפלט הבא:

```
<ROOT>
   <NODE_1 />
   <NODE_2 />
</ROOT>
```

באפשרותך להוסיף מאפיין לעץ XML או לשנות ערך של מאפיין על ידי השיטה **SetAttributeValue**, כמוצג בקוד זה:

```
static void Main(string[] args)
{
    XDocument xDoc  = new XDocument(
        new XElement("ROOT",
            new XAttribute("YEAR", "2008"),
            new XAttribute("TOTAL","2"),
            new XElement("NODE_01"),
            new XElement("NODE_02")
        )
    );
```

```
    XElement myRoot = xDoc .Element("ROOT");

    myRoot.SetAttributeValue("YEAR",2009);        // YEAR שנה מאפיין
    myRoot.SetAttributeValue("LANG", "CSharp");   // הוסף מאפיין חדש


    Console.WriteLine(xDoc );
}
```

קוד זה יפיק את הפלט הבא:

```
<ROOT YEAR="2009" TOTAL="2" LANG="CSharp">
   <NODE_01 />
   <NODE_02 />
</ROOT>
```

סוגי רכיבים נוספים

שלושה סוגים נוספים של רכיבים אשר שימשו בדוגמאות הקודמות הם **XComment**, **XDeclaration** ו-**XProcessingInstruction**. רכיבים אלה מוסברים בהמשך.

XComment

הערות XML מורכבות מטקסט בין היחידות הלקסיקליות <-- ו- --!>. תוכניות הפירוק (**parsers**) של XML מתעלמות מהטקסט שמופיע בין היחידות הלקסיקליות. תוכל להוסיף טקסט למסמכי XML באמצעות המחלקה XComment, כמוצג בשורת הקוד הבאה:

```
new XComment("this is a comment")
```

XDeclaration

מסמכי XML מתחילים בשורת קוד שכוללת את גרסת XML שנמצאת בשימוש, את סוג קידוד התווים (character encoding) ומציינת אם המסמך מסתמך על הפניות חיצוניות. חלק זה מכונה **הכרזת XML** (XML declaration), ושיבוצו נעשה על ידי המחלקה XDeclaration. הנה דוגמה למשפט XDeclaration:

```
new XDeclaration("1.0","utf-8","yes")
```

XProcessingInstruction

הוראות עיבוד של XML משמשות להעברת נתונים נוספים על אופן השימוש או הפירוק של מסמכי XML. השימוש השכיח ביותר בהוראות עיבוד הינו קישור של גיליון סגנון (style sheet) למסמכי XML.

תוכל לצרף הוראות עיבוד על ידי הבנאי XProcessingInstruction, אשר מקבל שתי מחרוזות בתור פרמטרים: **מחרוזת מטרה ומחרוזת נתונים**. כאשר הוראות העיבוד מקבלות פרמטרי

נתונים מרובים, חובה עליך לכלול את הפרמטרים הללו במחרוזת הפרמטר השני של הבנאי **XProcessingInstruction**, כפי שמוצג בקוד הבנאי שלהלן. שים לב שבדוגמה זו הפרמטר השני הוא מחרוזת מילולית, והגרשיים הכפולים שבתוך המחרוזת מיוצגים על ידי שני סימני גרשיים כפולים רצופים.

```
new XProcessingInstruction("xml-stylesheet",
               @"href=""stories.css"", type=""text/css""")
```

בקוד הבא נעשה שימוש בכל שלושת הבנאים:

```
static void Main(string[] args)
{
    XDocument xDoc = new XDocument(
        new XDeclaration("1.0","utf-8","yes"),
        new XComment("This is a comment"),
        new XProcessingInstruction("xml-stylesheet",
                   @"href=""stories.css"", type=""text/css""),

        new XElement("ROOT",
            new XElement("NODE_01"),
            new XElement("NODE_02")
        )
    );
}
```

קוד זה יפיק את הפלט הבא בקובץ הפלט. אולם, שימוש בשיטה WriteLine של xd לא יגרום להצגת משפט ההכרזה, למרות שהוא מהווה חלק מקובץ המסמך:

```
<? xml version="1.0" encoding="utf-8" standalone="yes" ?>
<!--This is a comment -->
<? xml stylesheet href="stories.css" type="text/css" ?>
<ROOT >
   <NODE_01 />
   <NODE_02 />
</ROOT>
```

שימוש בשאילתות LINQ ל–XML

ניתן לצרף ביטוי שאילתה **XML LINQ API**, כדי לבצע חיפושים פשוטים אך יעילים ביותר בעץ XML.

הקוד שלהלן יוצר עץ XML פשוט, מציג אותו על המסך ושומר אותו בקובץ MyLinqSample.xml. קוד זה אינו מחדש דבר, אולם העץ שהוא יוצר ישמש אותנו בדוגמאות הבאות.

```
static void Main(string[] args)
{
    XDocument xd = new XDocument(
        new XElement("MyElements",
            new XElement("first",
                new XAttribute("color","red"),
                new XAttribute("size", "small")),
            new XElement("second",
                new XAttribute("color", "red"),
                new XAttribute("size", "medium")),
            new XElement("third",
                new XAttribute("color", "blue"),
                new XAttribute("size", "large"))));

    Console.WriteLine(xd);
    xd.Save("MyLinqSample.xml");
}
```

קוד זה יפיק את הפלט הבא:

```
<MyElements >
   <first color="red" size="small"  />
   <second color="red" size="medium"  />
   <third color="blue" size="large"  />
</ MyElements >
```

בקוד הבא פעילה שאילתת LINQ פשוטה לבחירת סדרת משנה של רכיבים מתוך עץ XML והצגתם במספר דרכים שונות. קוד זה מבצע את הפעולות הבאות:

- בוחר מעץ XML רק את האלמנטים שאורך השם שלהם 5 תווים. שמות האלמנטים הם first, second ו-third. first ו-third הם שמות הרכיבים היחידים שמתאימים לקריטריון החיפוש, ולכן רק הם ייבחרו.
- מציג את שמות האלמנטים שנבחרו.
- מסדר ומציג את הרכיבים שנבחרו, כולל שמות הרכיבים וערכי המאפיינים. שים לב שהמאפיינים נשלפים באמצעות השיטה **Attribute**, וערכי המאפיינים נשלפים עם המאפיין **Value**.

```
static void Main(string[] args)
{
  XDocument xd = XDocument.Load("MyLinqSample.xml");   // טען את המסמך
  XElement myRoot = xd.Element("MyElements");          // קבל את השורש

  // בחר אלמנטים בעלי אורך של חמש אותיות
  var fiveLetterWords = from e in myRoot.Elements()
                        where e.Name.ToString().Length == 5
                        select e;
```

```
    foreach (XElement x in fiveLetterWords)
        Console.WriteLine(x.Name.ToString());

    Console.WriteLine();
    foreach (XElement x in fiveLetterWords)
        Console.WriteLine("Name: {0}, color: {1}, size: {2}",
                          x.Name,
                          x.Attribute("color").Value,
                          x.Attribute("size").Value);
}
```

קוד זה יפיק את הפלט הבא:

```
first
third

Name: first, color: red, size: small
Name: third, color: blue, size: large
```

הקוד הבא מפעיל שאילתה פשוטה לשליפת כל האלמנטים העליונים (top-level) בעץ XML, ויוצר אובייקט מסוג אנונימי עבור כל אחד מהם. השימוש הראשון של השיטה WriteLine נועד להצגת סידור ברירת המחדל של הסוג אנונימי. משפט **WriteLine** השני מסדר באופן מפורש את חברי האובייקטים מהסוג אנונימי.

```
static void Main(string[] args)
{
  XDocument xd = XDocument.Load("MyLinqSample.xml");   // טען את המסמך
  XElement myRoot = xd.Element("MyElements");          // קבל את השורש

  var values = from e in myRoot.Elements()
               select new { e.Name, color = e.Attribute("color")
};


  foreach (var x in values)
      Console.WriteLine(x);

  Console.WriteLine();
  foreach (var x in values)
      Console.WriteLine("{0, -6}, color: {1, -7}", x.Name,
                                                   x.color.Value);
}
```

קוד זה יפיק את הפלט הבא. בשלוש השורות הראשונה מוצג סידור ברירת המחדל של הסוג האנונימי. בשלוש השורות האחרונות מוצג הסידור המפורש שמופיע במחרוזת הסידור בקריאה השנייה של השיטה WriteLine:

```
{ Name = first, color = color="red" }
{ Name = second, color = color="red" }
{ Name = third, color = color="blue" }

first , color: red
second, color: red
third , color: blue
```

דוגמאות אלו ממחישות כמה קל לשלב את **XML LINQ API** עם כלי השאילתה של LINQ ליצירת יכולות יעילות ביותר לחקירת עץ XML.

סיכום

בפרק זה הכרנו את LINQ to XML, למדנו להשתמש במחלקות החדשות כדי לבנות ולעדכן מסמכי XML, סקרנו את המאפיינים החדשים אשר מקלים על העבודה עם XML, כולל שימוש בממשק השאילתה של LINQ לתחקור מסמכי XML.

אינדקס

האינדקס באנגלית והוא מתחיל מסוף הספר בעמוד 1 ומתקדם פנימה אל תוך הספר.

האינדקס לפרקים 29-32 נמצא בעמוד הבא.

אינדקס לפרקים 29-32

W

T

Q

R

N

F

E

C

B

Index

ת.ד. 6108 הרצליה 46160 **הזמנות באתר**
info@hod-ami.co.il www.**hod-ami**.co.il

קטלוג ינואר 2017

	כולל	עמ'	מחיר*
אינטרנט - מפתחי אתרים/גרפיקה			
אמא, אבא - בניתי אתר באינטרנט (HTML)		256	29
הגדל את הכנסות העסק שלך באמצעות **פרסום בגוגל Google AdWords**		300	249
HTML5 המדריך לבניית אתרים ולמערכות WEB, הדור הבא – מהד' 2		390	159
The **Java** Tutorial סדנת לימוד	CD	768	179
JavaScript סדנת לימוד		586	159
מדריך **ASP.NET MVC 4**		514	199
ASP.NET 3.5 סדנת לימוד בשפות C# ו-VB		824	99
תכנות			
Code Complete – מדריך מעשי לפיתוח תוכנה		288	139
לחפש באגים, מדריך מעשי לבודק תוכנה, מהד' 3		350	169
Visual C# 3.0 סדנת לימוד		656	99
ללמוד **C** - מהד' 3		480	139
שפת **אסמבלי** למחשב האישי, מהד' 2		314	89
יסודות התכנות ב-**VBA** לתוכנת **Excel**, מהד' 4		152	95
PC - חומרה, תוכנה ורשתות			
מדריך Hacking ואבטחת מידע, מהד' 2		428	169
מדריך **חומרה ותוכנה** לטכנאי PC - מהד' 5 (כולל חלונות 7/8)		752	189
מדריך **רשתות** לטכנאי PC ולמנהלי רשת - מהד' 4		608	219
Windows			
Windows 8.1 מדריך למשתמש		544	149
Windows 8 מדריך למשתמש		438	19
Windows 7 צעד-אחר-צעד		272	19
LINUX			
LINUX למתקדמים, טיפים, טריקים ותכנות ב-BASH		226	189

* מחיר מומלץ לצרכן כולל מע"מ

היכנס לאתר להתעדכן בספרים החדשים ובמחירי המבצע בהוצאה

תוכן עניינים ופרקים לדוגמה www.**hod-ami**.co.il

	כולל	עמ'	מחיר*
גרפיקה			
Flash – ספר הדרכה ותרגילים		122	64
אינדיזיין – ספר הדרכה ותרגילים		132	72
Illusatrator – ספר הדרכה ותרגילים		120	64
Photoshop צעד אחר צעד (צבע מלא, למתחילים), מהד' 3 כריכה קשה		200	159
Photoshop צעד אחר צעד (ש/ל, למתחילים), מהד' 3		200	89
מדריך לתוכנת העיצוב והאנימציה **3ds max** (2 כרכים)	CD	1400	289
OFFICE			
יישומי סטטיסטיקה בגיליון אלקטרוני Excel		86	69
סטטיסטיקה יישומית		150	97
טבלאות ציר – ניתוח נתונים חכם		116	87
יסודות התכנות ב-**VBA** לתוכנת **Excel**, מהד' 4		152	95
Access 2016 צעד אחר צעד חדש		384	169
Word 2016 צעד אחר צעד		150	59
Excel 2016 צעד אחר צעד		336	129
PowerPoint 2010 צעד אחר צעד		202	69
Outlook 2010 צעד אחר צעד		190	69
Access 2010 סדנת לימוד		360	179
עוד ספרים בגרסאות קודמות (2007 ו-2003) ניתן למצוא באתר הוד-עמי			
ניהול, כלכלה ושונות			
לחפש באגים, מדריך מעשי לבודק תוכנה, מהד' 3		350	169
ניהול ממוקד לעשות יותר עם מה שיש (כריכה קשה) - מהד' 4		358	133
לי זה עולה יותר (תמחיר) (כריכה קשה) - מהד' 3		368	129
מערכות מידע			
Oracle SQL יכולות מתקדמות		626	249
SAS (**Statistical Analysis System**) – ספר לימוד		256	169
בסיסי נתונים ושפת **SQL** – עקרונות ועיצוב		648	149
ניתוח מערכות מידע כולל מתודולוגיית ה- **UML**		818	229
המדריך העברי השלם **UML**		346	329
ספרים דיגיטליים			
לרכישת ספרים לצפייה בפורמט PDF היכנס לאתר לקטגוריה "ספרים דיגיטליים"			
קבצי תרגול לספרים			
קבצי תרגול לספרים שונים תמצא באתר בקטגוריה "קבצי תרגול לספרים"			

* מחיר מומלץ לצרכן כולל מע"מ. קטלוג 1/2017

היכנס לאתר להתעדכן בספרים החדשים ובמחירי המבצע בהוצאה

תוכן עניינים ופרקים לדוגמה www.**hod-ami**.co.il