C#

למתכנתי
Java/C++/Visual C++

יש להתעלם מכל מה שקשור לתקליטור המצורף

הקבצים נמצאים באתר הוד-עמי

www.hod-ami.co.il

בקטגוריה "קבצי תרגול לספרים" יש לבחור

ב-59327 כדי להוריד את הקבצים למחשב

עורך ראשי, עריכה מקצועית: **זהר עמיהוד**

תרגום: **איציק מנשרוף**

ייעוץ מקצועי: **ארז קורן**

עריכה לשונית ועיצוב: **רמה שנקלר**

עיצוב עטיפה: **שרון רז**

- **טלפון: 09-9564716**
- **פקס: 09-9571582**
- **דואר אלקטרוני: info@hod-ami.co.il**
- **אתר באינטרנט: www.hod-ami.co.il**

C#

למתכנתי
Java/C++/Visual C++

Tom Archer

Microsoft®

הוצאת
הוד-עמי
ספרי מחשבים

Inside C#

By Tom Archer

Editor: **Z. Amihud**

הוצאת הוד-עמי
לספרי מחשבים בע"מ
ת.ד. 6108 הרצליה 46160
טלפון: 09-9564716 פקס: 09-9571582
info@hod-ami.co.il

ISBN 965-361-296-4 מסת"ב

תוכן עניינים מקוצר

תוכן עניינים

C# למתכנתי JAVA/C++

*מאת **Scott Wiltamuth**, חבר בצוות*
הפיתוח של C# בחברת Microsoft.

כל שנות עבודתי במיקרוסופט הושקעו בניסיון לשפר את יכולתו של מפתח התוכנה, ובעיקר להגדיל את יצרנותו. עבודתי הקיפה מיגוון רחב של מוצרים וטכנולוגיות, אך מעולם לא הייתי כה נלהב מעבודתי כפי שאני עכשיו. הטכנולוגיות שמספקת Microsoft .NET מדהימים ברוחבם ובעומקם. אנו מספקים שפה חדשה ונהדרת, השוברת את כל המחסומים אשר באופן מסורתי חילקו את המפתחים לעולמות שונים ובלתי-שקולים, ומאפשרת שיתוף פעולה בין אתרי אינטרנט כדי לענות על צרכי המשתמשים.

הבה נסקור את אבני היסוד של NET ומספר טכנולוגיות קשורות:

- שפת C#, שפה חדשה – C# היא השפה הראשונה מוכוונת הרכיבים (component oriented) במשפחת השפות C ו-C++. שפה זו, הנגזרת מ-C ו-C++, פשוטה, מודרנית, מוכוונת עצמים ומאפשרת לעבוד עם סוגי נתונים מורשים בלבד. C# משלבת את היצרנות הגבוהה של Microsoft Visual Basic עם העוצמה הגלומה ב- C++.
- **Common Language Runtime** – סביבת CLR עם ביצועים מעולים, הכוללת מנוע ביצוע, garbage collector (אוסף האשפה), הידור במועד Just-In-Time, מערכת אבטחה ו- NET Framework.
- **Common Language Specification** – תקן CLS מתאר רמה משותפת של פונקציונליות עבור שפת תכנות. החסם המינימלי יחסית של CLS מאפשר יצירת "מועדון" של שפות תואמות – CLS. כל אחד מחברי המועדון נהנה משני יתרונות: גישה מלאה לפונקציונליות של מסגרת NET ותפעוליות שיתופית עשירה עם שפות תואמות אחרות. לדוגמה, מחלקה של Visual Basic יכולה לרשת ממחלקה של C# ולדרוס את השיטות הווירטואליות שלה.
- **אוסף שפות עשיר המיועד ל-CLR** – מיקרוסופט מספקת שפות העומדות בדרישות CLS, כמו Visual Basic, Visual C++ עם הרחבות, C# ו-JScript. בנוסף לאלו קיימים שפות צד ג' אחרות רבות מכדי לפרטן כאן.

- **שירותי אינטרנט (Web)** – רשת האינטרנט מורכבת כיום בעיקר מאתרים נפרדים. כאשר משתמש מבקר באתרים מרובים כדי לבצע משימה נתונה, כגון סידורי נסיעה עבור קבוצת אנשים, אתרים אלה אינם משתפים פעולה ביניהם. הדור הבא של האתרים יתבסס על שיתוף רשתות של אתרי האינטרנט. הסיבה לכך פשוטה: שיתופיות בין אתרי אינטרנט יכולה לפעול טוב יותר כדי לענות על צרכי המשתמשים. טכנולוגיות שירותי אינטרנט מעודדות שיתוף פעולה בין אתרים בכך שהן מאפשרות תקשורת באמצעות פרוטוקולים מבוססי-XML סטנדרטיים שהם בלתי-תלויים בשפה ובלתי-תלויים בפלטפורמה גם יחד. שירותי אינטרנט רבים יתבססו על C# ועל CLR המתבצעים על Windows, אך אפשרויות הארכיטקטורה אינן מוגבלות.
- **Visual Studio .NET** – קושר יחד את כל המרכיבים ומאפשר ליצור בקלות מיגוון רחב של רכיבים, יישומים ושירותים, בשפות תכנות שונות.

לאחר שנגעתי במספר טכנולוגיות חשובות הקשורות ל-C#, הבה נסקור את C# עצמה. מפתחים השקיעו ומשקיעים רבות בשפה שבה בחרו, ולכן מחובתנו להוכיח את כדאיותה של שפה חדשה. נעשה זאת באמצעות שילוב מסוים בין שימור פשוט, שיפורים מתקנים וחדשנות מעמיקה.

שימור פשוט

הפילוסוף היווני היפוקרטס אמר "עשה לך הרגל של שני דברים: לעזור, או לפחות לא להזיק". החלק של "לפחות לא להזיק" שיחק תפקיד חשוב בתכנון של C#. אם C או C++ טיפלו בבעיה היטב, יכולת זו נשמרה ללא שינוי. חשוב מכך, C# מעתיקה מ-C ו-C++ ביטויים, משפטים וגם את התחביר הכללי. מכיון שחלק נכבד של כל תוכנית טיפוסית מכיל את כל התכונות האלו, מתכנתי C ו-C++ לא יתקשו בעבודה עם C#.

שיפורים מתקנים

ב-C# נעשו שיפורים רבים – יותר מדי מכדי להזכירם בהקדמה קצרה זו – אך מן הראוי לציין מספר שינויים שבאים למנוע שגיאות רגילות ושגיאות זוללות זמן של C ו-C++:

- חובה להכריז על משתנים לפני שמשתמשים בהם, כדי למנוע שגיאות הנגרמות ממשתנים לא מאותחלים.
- התנאים במשפטים כמו if ו-while דורשים ערכים בוליאניים, כך שמפתח המשתמש בטעות באופרטור ההצבה (=) במקום באופרטור ההשוואה (==) מאתר את השגיאה בזמן ההידור.
- כישלון שקט (silent fall-through) במשפטי switch נמנע; כך, מתכנת המשמיט בטעות משפט break יכול לאתר את השגיאה בזמן ההידור.

חדשנות מעמיקה

חדשנות מעמיקה נמצאת במערכת הסוגים המשותפים (CTS - Common Type System) של #C, וכוללת את החידושים הבאים:

- מערכת הסוגים של #C מיישמת ניהול זיכרון אוטומטי, ובכך משחררת את המפתחים מניהול ידני צורך-זמן ומועד לשגיאות. שלא כמרבית מערכות הסוגים האחרות, מערכת הסוגים של #C גם מאפשרת מניפולציה על סוגי מצביעים ועל כתובות של אובייקטים (טכניקות ניהול זיכרון ידניות אלו מורשות רק בהקשרי אבטחה מסוימים).
- מערכת הסוגים של #C הינה עקבית – כל דבר הוא אובייקט. #C מגשרת על הפער בין סוגי ערכים (value types) וסוגי ייחוס (reference types) באמצעות שימוש במושגים מרכזיים וחדשניים הנקראים boxing (אריזה) ו-unboxing (פריקה). כך היא מאפשרת להתייחס אל כל פיסת נתונים כאל אובייקט.
- מאפיינים (properties), שיטות (methods) ואירועים (events) הם מרכיבים חיוניים של השפה. שפות רבות אחרות חסרות תמיכה פנימית עבור מאפיינים ואירועים, ובכך יוצרות חוסר התאמה מיותרת בין השפה לבין המערכות. לדוגמה, אם המערכת תומכת במאפיינים והשפה אינה עושה זאת, הגדלת ערך המאפיין הופכת להיות מסורבלת, לדוגמה, o.SetValue(o.GetValue()+1). אם גם השפה תומכת במאפיינים, הפעולה היא פשוטה, o.Value++.
- #C תומכת בתכונות (attributes) המאפשרות להגדיר ולהשתמש במידע הצהרתי על רכיבים. היכולת להגדיר סוגים חדשים של מידע הצהרתי היתה תמיד כלי רב-עוצמה עבור מתכנני השפה. יכולת זו נמצאת כעת ברשותם של כל מפתחי #C.

על הספר

המחבר Tom Archer בונה את תשתית הלימוד על ידי הצגת NET ו-CLR. הוא מסביר את מרכיבי היסוד של #C ויורד לעומקם של מספר מושגים מתקדמים בשפה זו. ניסיונו הרב – הן כמפתח והן כמחבר ספרים על C++, J++ ו- Microsoft Windows – מאפשר לו להסביר את #C בדרכים מהנות ומאלפות.

קורא יקר, אני מקווה שתיהנה לכתוב את תוכניות #C הראשונות שלך, שתהיינה הראשונות בתוכניות רבות שתכתוב בעתיד.

Scott Wiltamuth
C# Design Team Member
Microsoft Corporation

הערה מפי העורך:

Scott Wiltamuth, עובד Microsoft, טוען (ויש צדק בטענותיו) שמתכנתי C/C++ ירגישו בנוח בעבודה עם #C. לדעתי (הצנועה) מתכנתי Java ירגישו נוח יותר. לא לחינם קוראים ל- #C Microsoft בשם Java Killer.

הקדמה

מדוע כתבתי את הספר

כמפתח תוכנה במשך עשרים שנה – אני חש מבוגר יותר בכל פעם שאני חושב על כך – הגעתי לנקודה שבה תכנות החל להיות מעט נדוש עבורי. אל תבינו אותי לא נכון: אם הייתי מולטי מיליונר ולא הייתי צריך לעבוד, קרוב לוודאי שהייתי ממשיך בכתיבת קוד, כי זה דבר שבאמת אני נהנה לעשותו. עם זאת, הגעתי לנקודה שבה חשבתי לעצמי, "הכל כבר נעשה!". אז הופיעו Microsoft .NET ו-C#, ועולם חדש ושלם של אפשרויות נפתח בפניי. שוחחתי עם מספר חברים שחוו את אותה תחושה של "התעוררות מחדש" עם הצגת .NET לראשונה. יש בידינו את הטכנולוגיה החדשה והמלהיבה הזאת, אשר סוף סוף מתמודדת עם סוגיות שטיפלנו בהן במשך שנים (לדוגמה, סביבות פיתוח מרובות שפות, בעיות פיתוח וניהול גרסאות של מערכות גדולות ומורכבות, וכן הלאה). כתבתי ספר זה כי החזרה לכתיבת קוד מרתקת אותי, ושוב אני מתעורר בכל בוקר וחושב על הדבר החדש והמלהיב שאלמד היום. אני מקווה כי במהלך לימוד השפה תהיה גם אתה שותף להתלהבות זו.

נכון לזמן כתיבת הספר, כל הכותב ספר על C# לומד למעשה את השפה בזמן הכתיבה. אם כתבת יישום במהלך לימוד SDK או C#, אתה יודע שזו עשויה להיות חוויה לא קלה. כעת נסה לדמיין שאתה כותב משהו שעשרות אלפי אנשים עומדים לקרוא לאחר שסיימת! הבעיה הגדולה היא שבמחצית הדרך – ברגע שהבנת מה אתה עושה – אתה מתחבט בתכנון וכתיבה מחדש של הכל! כמובן שאין זה מעשי כאשר קיימים תאריכי יעד סופיים. אני מאמין שספר זה משמש כלי טוב ללימוד C#. היות ולמדתי תוך כדי כתיבתו, עשויה להיות מידה מסוימת של חוסר עקביות ועשויים להיות מספר דברים שניתן היה לשפר אותם בהרבה. עם זאת, אם תהיה לי הזדמנות לכתוב מהדורה שנייה, אני יכול להבטיח שגם אתה תוכל להפיק תועלת מעקומת הלמידה האישית שלי.

שגיאות

אני יצור אנושי שלפעמים עושה טעויות. אני מכיר בזה ומוכן לקבל בברכה כל משוב בנוגע לספר שכתבתי. איני מאלה האומרים "אני כל כך טוב כי אני כותב ספרים". אני רק בחור רגיל שהתמזל מזלו וקיבל הזדמנות לכתוב ספר. אני תמיד פתוח ללמוד מאחרים, ולמעשה אוהב לעשות כך. ניתן להגיע אלי בכתובת http://www.TheCodeChannel.com. באתר תוכלו למצוא תחת הכותרת Errata את רשימת הדברים שיש לתקן בספר.

הערת העורך: במהדורה העברית **תוקנו** כל ההערות הידועות נכון ליום הוצאת הספר לאור. לא כן במהדורה האלקטרונית המלאה של הספר באנגלית המצורף לתקליטור. בכל מקרה, אני ממליץ לכל אחד להתעדכן בדף Errata באתר המחבר.

למי מיועד הספר

ספר זה מיועד למי שרוצה להתחיל ללמוד פיתוח #C ו-NET. כפי שציינתי, זו פלטפורמה מרתקת עם עתיד מבטיח בכל הנוגע לפיתוח מבוזר עבור Microsoft Windows. ספר זה מניח שיש לך רקע באחת מהשפות C, ++C, ++Visual C או Java. אני סבור שהתנאי המוקדם הנוסף הוא רק הרצון ללמוד ולגלות מימדים חדשים בכתיבת יישומים. מכיון שאתה אוחז בספר, כנראה שיש לך את הרצון הזה.

מבנה הספר

ספר זה אורגן בקפידה כרצף לוגי של מספר חלקים; כל חלק מכיל קבוצת פרקים שעוסקים בנושא מרכזי של פיתוח #C או NET.

הספר מתחיל בחלק 1 – "הנחת היסודות", המיועד למתכנתי #C מתחילים ולאלה שסביבת NET חדשה עבורם. הפרקים בחלק זה מספקים הקדמה ל-OOP וסביבת NET ומדגימים איך ליצור ולבדוק את תוכניות #C הראשונות.

חלק 2 – "יסודות המחלקה של #C", מציג את היסודות של הגדרה ועבודה עם מחלקות (classes) ב-#C. הפרקים בחלק זה מכוונים לספק לך בסיס מוצק אודות החברים הנתמכים ב-#C (enums, properties, arrays וכן הלאה) ואיך להגדירם ולהשתמש בהם ביישום #C.

למרות שבפרקים קודמים כתבת קוד במונחים של הגדרת חברי מחלקה (class members), בחלק 3 – "כתיבת קוד", תתחיל לבחון היבטים אחרים של משימות כמו זרימת תוכנית (program flow), טיפול בשגיאות עם חריגים (exception) וכתיבת מטפלים באירועים (event handlers) באמצעות נציגים (delegates).

חלק 4 – "נושאים מתקדמים ב-#C", מסיים את הספר. בהיותי "מכור מחשבים" טיפוסי, נהניתי מאד לכתוב אותו. חלק זה מכיל פרקים על תכנות מרובה מטלות (multithreading), שיקוף (reflection), עבודה עם קוד לא-מנוהל (unmanaged code) הכולל תפעוליות שיתופית עם רכיבי COM, והבחנת גרסאות (versioning).

אודות התקליטור המצורף

לספר זה מצורף תקליטור. ראה נספח בסוף הספר – התקליטור המצורף – להוראות התקנה והפעלה של התקליטור.

אם במחשב שלך מסומנת האפשרות AutoRun (הפעלה אוטומטית של תקליטור) הקיימת ב-Windows, יופיע חלון פתיחה כאשר תכניס את התקליטור לכונן התקליטורים ויוצגו מספר אפשרויות התקנה. בכדי לפתוח חלון זה באופן ידני הפעל את התוכנית **Hodami.exe** מספריית השורש של התקליטור.

ספר אלקטרוני eBook

את הגירסה המלאה האלקטרונית של הספר באנגלית תוכל למצוא בתקליטור, בתיקיה **59327\eBook**. אם ביצעת התקנה לדיסק במחשב שלך, תוכל למצוא אותה באותה תיקיה. אתר והפעל את קובץ **setup.exe**.

קוד מקור

תוכניות הדוגמה של הספר נמצאות בתקליטור המצורף, בתיקיה **59327**. תוכל לעיין בדוגמאות ישירות מהתקליטור, או להתקינן בדיסק באמצעות תוכנית ההתקנה הניתנת להפעלה דרך **Hodami.exe**.

דרישות מערכת

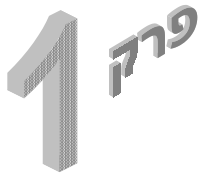
כדי להפיק את המירב מספר זה, מומלץ מאוד לעיין ביישומי הדוגמה במהלך הקריאה של כל פרק. לפיכך יש להתקין את הגירסה האחרונה של .NET Framework SDK. בזמן כתיבת שורות אלו, ערכת הפיתוח כוללת את סביבת הריצה של .NET ואת מהדר C#. נוסף לכך, נמנעתי במתכוון להשתמש במוצר Visual Studio .NET בכדי להתמקד בשפה ובסביבת הריצה, ולא להציב מגבלות כלשהן על סביבת הפיתוח המסוימת שלך. משום כך, כל הדוגמאות בספר ניתנות להידור וביצוע משורת הפקודה.

על היועץ המקצועי

ארז קורן, בוגר יחידת המחשב בצה"ל – ממר"ם, מתמחה בעולם האינטרנט, e-Commerce וטכנולוגיות חדשניות. בעל ניסיון עשיר בפיתוח ביישומי C++ ו-Java ליישומים עתירי פלטפורמות ומשתמשים. כיום, מנהל צוות פיתוח בחברת Comverse בתחום המתפתח של המסחר האלקטרוני.

חלק 1

הנחת היסודות



יסודות התכנות מכוון-עצמים

פרק זה יצעיד אותך אל עולם המושגים של **תכנות מכוון-עצמים** – **OOP** (Object-Oriented Programming), או **מכוון אובייקטים**, ויעזור לך להבין את החשיבות של עקרונות אלה לתכנות. על שפות תכנות רבות, כגון C++ או Visual Basic, נאמר שהן "תומכות באובייקטים", אך למעשה רק שפות מעטות תומכות בכל העקרונות שתכנות מכוון-עצמים מושתת עליהם. C# היא אחת מהשפות האלו: היא תוכננה מיסודה להיות שפה מכוונת-עצמים ומבוססת-רכיבים אמיתית. לכן, כדי להפיק את מלוא התועלת מספר זה, עליך להבין היטב את העקרונות שיוצגו כאן.

קוראים המבקשים "לצלול" ישר לתוך הקוד, נוהגים בדרך כלל לדלג על פרקים העוסקים במושגים ועקרונות. עם זאת, מומלץ שתקרא פרק זה בעיון, אלא אם אתה מומחה בנושא. גם אם יש לך מושג כלשהו בתכנות מכוון-עצמים, תוכל להפיק תועלת רבה מקריאת הפרק. זכור שהפרקים הבאים יתייחסו למונחים ולעקרונות הנידונים כאן, ולכן בכל מקרה כדאי לסקור את תוכנו.

שפות רבות אמורות להיות מכוונות-עצמים או מבוססות-עצמים (אובייקטים), אך רק מעטות הן כאלו באמת. שפת C++ אינה כזו, כי שורשיה נעוצים בשפת C, עובדה שלא ניתן לשנות. אידיאלים רבים מאוד של תכנות מכוון-עצמים הוקרבו בשפת C++ לטובת תמיכתה בקוד הקיים שנכתב בשפת C. אפילו לשפת Java, טובה ככל שתהיה, יש מספר מגבלות כשפה מכוונת-עצמים. ליתר דיוק, אתייחס לעובדה שבשפה זו קיימים סוגי אובייקטים וסוגי נתונים בסיסיים המטופלים ומתנהגים באופן שונה מאוד זה מזה. עם זאת, פרק זה אינו מתמקד בהשוואת הנאמנות של שפות שונות לעקרונות של OOP, אלא מציג לימוד אובייקטיבי של עקרונות אלה.

לפני שנתחיל, ברצוני להוסיף שתכנות מוכוון-עצמים הוא הרבה יותר מצירוף מילים מסחרי (למרות שהפך לכזה עבור מספר אנשים), הוא יותר מתחביר חדש, או מממשק תכנות יישומים (API) חדש. תכנות מוכוון-עצמים הוא מערכת שלמה של עקרונות ורעיונות, המייצגים דרך חשיבה לטיפול ולהתמודדות אינטואיטיביים בתוכנית מחשב, ולפיכך זהו תכנות יעיל יותר.

עבודתי הראשונה היתה כרוכה בכתיבת יישום מסחרי בשפת Pascal. בהמשך כתבתי יישומים ב-PL/I וב- RPG III (וגם ב-RPG/400). כעבור מספר שנים התחלתי לכתוב יישומים בשפת C. בכל אחד מהמקרים יכולתי ליישם בקלות את הידע שצברתי מהניסיון הקודם. עקומת הלמידה התקצרה במעבר משפה לשפה ללא תלות במורכבות השפה שלמדתי. הסיבה לכך היא שעד שהתחלתי לכתוב תוכניות ב- ++C, כל השפות שבהן השתמשתי היו שפות פרוצדורליות הנבדלות זו מזו בעיקר בתחביר.

אם תכנות מוכוון-עצמים חדש לך, עלי להזהירך: ניסיון קודם עם שפות שאינן מוכוונות-עצמים לא יועיל לך כאן! תכנות מוכוון-עצמים הוא דרך חשיבה שונה על אופן התכנון והתכנות של פתרונות לבעיות. למעשה, מחקרים מוכיחים שמתכנתים מתחילים לומדים שפות מוכוונות-עצמים הרבה יותר מהר מאלה אשר החלו בשפות פרוצדורליות כמו BASIC, COBOL ו-C. כל אלה אינם צריכים "להשתחרר" מהרגלים פרוצדורליים ולשנות דפוסי חשיבה שעלולים להקשות על הבנת OOP, אלא מתחילים מהבסיס. אם כתבת תוכניות בשפות פרוצדורליות במשך שנים רבות ו-#C היא השפה מוכוונת-העצמים הראשונה שלך, העצה הטובה ביותר שאוכל לתת לך היא לשמור על ראש פתוח כדי ליישם את הרעיונות שאציג כאן בטרם תיכנע לטובת פתרון לא אמיתי לבעיה בשפה פרוצדורלית כלשהי. כל מי שמגיע לתכנות מוכוון-עצמים מרקע של שפה פרוצדורלית עובר תהליך המתבטא בעקומת למידה, ואשר משתלם לו בסיום התהליך. התועלת בכתיבת תוכניות בשפה מוכוונת-עצמים רבה מאוד, הן במונחים של תכנות יעיל יותר והן בבניית מערכת שניתן לשנותה ולהרחיבה בקלות מרגע שנכתבה. אולי בהתחלה אין זה נראה כך, אך במהלך כ-20 שנה של פיתוח תוכנה (כולל 8 האחרונות בשפות מוכוונות-עצמים) למדתי שעקרונות OOP אכן עונים על הציפיות מהם, **אם הם מיושמים באופן נכון**.

כל דבר הוא אובייקט

בשפה מוכוונת עצמים אמיתית, כל היישויות של **מרחב הבעיה** (problem domain) מבוטאות באמצעות הרעיון של **אובייקטים** (שים לב שבספר זה אשתמש בהגדרה של Coad/Yourdon ל"מרחב הבעיה" – שמתאר את הבעיה שמנסים לפתור, במונחים המייחדים אותה, כמו מורכבות, מושגים, אתגרים וכו'). כפי שאולי ניחשת, השימוש ב**אובייקטים** (objects) הוא הרעיון המרכזי שעומד מאחורי תכנות מוכוון-עצמים. בדרך כלל אין אנו חושבים במונחים של מבנים, חבילות נתונים, קריאות לפונקציות ומצביעים, אלא במונחים של אובייקטים. הנה דוגמה הממחישה זאת.

אם היית כותב יישום להפקת חשבוניות והיית צריך לסכם את שורות הפירוט של החשבונית, איזו גישת מחשבה מבין הבאות היתה אינטואיטיבית יותר מנקודת המבט של הלקוח?

- **גישה שאינה מוכוונת-עצמים** - תהיה לי גישה למבנה נתונים המייצג את כותרת החשבונית. מבנה כותרת החשבונית יכלול גם רשימה מקושרת של מבנים המייצגים את פירוט החשבונית, כאשר כל מבנה מכיל ערך שורת סכום. לכן, כדי לקבל את הסכום הכולל של החשבונית, עלי להצהיר על משתנה הנקרא למשל totalInvoiceAmount, לאתחל אותו בערך 0, לקבל מצביע למבנה כותרת החשבונית, לקבל את ראש הרשימה המקושרת של שורות הפירוט, ואז לנוע על פני הרשימה. בכל קריאה של מבנה שורת פירוט, אקבל את **משתנה המחלקה** (member variable) שלו המכיל את הסיכום עבור אותה שורה, ואוסיף אותו למשתנה totalInvoiceAmount.

- **גישה מוכוונת-עצמים** - יהיה לי אובייקט חשבונית, ואשלח לו הודעה המבקשת ממנו את הסכום הכולל של החשבונית. איני צריך לחשוב על האופן שבו האובייקט שומר את המידע הפנימי שלו, כפי שהיה עלי לעשות עם מבנה הנתונים בגישה הקודמת. אני פשוט נוהג עם האובייקט בדרך טבעית של הפניית בקשות באמצעות שליחת הודעות (אוסף הבקשות שאותן האובייקט מסוגל לעבד נקרא בשם כללי **ממשק האובייקט** (object's interface). בהמשך אסביר מדוע חשיבה במונחים של **ממשק** (interface) ולא של **מימוש** (implementation), או יישום, מוצדקים בגישה מוכוונת-עצמים, כפי שעשיתי כאן).

אין ספק שגישה מוכוונת-עצמים היא אינטואיטיבית וקרובה יותר לדרך החשיבה של רוב האנשים המטפלים בבעיה העומדת בפניהם. בפתרון השני (מוכוון-עצמים), סביר שאובייקט החשבונית מבצע איטרציה על **אוסף** (collection) אובייקטים של פרטי חשבונית ושולח לכל אחד מהם הודעה המבקשת לקבל את שורת הסיכום שלו.

אולם, אם אתה מבקש סיכום כולל, **אינך צריך לטפל באופן הביצוע של התהליך**. אינך דואג לכך, כי אחד מעיקרי תכנות מוכוון-עצמים הוא **כימוס** (encapsulation) – היכולת של אובייקט להסתיר את השיטות והנתונים הפנימיים שלו ולהציג ממשק המאפשר לתוכנית לגשת רק לחלקיו החשובים. אין חשיבות לדרך הפנימית שבה האובייקט מבצע את עבודתו, כל זמן שהוא מסוגל לבצע עבודה זו. הוא מציג ממשק שבאמצעותו אתה גורם לו לבצע עבודה מוגדרת ומועילה עבורך (בהמשך הפרק אסביר ביתר פירוט את העקרונות של כימוס וממשקים). הנקודה החשובה היא, שקל מאוד לתכנן ולכתוב תוכניות מחשב המחקות את האובייקטים של מרחב הבעיה בעולם הממשי, כי הן מאפשרות לנו לחשוב בדרך טבעית יותר.

שים לב שגישת העיבוד מוכוון-עצמים דורשת אובייקט שיבצע עבודה מועילה עבורך – כלומר, סיכום של כל שורות הפירוט. שים לב שאובייקט אינו מורכב מנתונים בלבד, כמו שקיים במבנה נתונים. על פי ההגדרה, אובייקטים כוללים נתונים ושיטות הפועלות על נתונים אלה. כלומר, כאשר מתייחסים למרחב בעיה, ניתן לעשות יותר מאשר תכנון של מבני נתונים הדרושים לבעיה. ניתן גם לבחון אילו שיטות יש לשייך לאובייקט נתון ובכך ליצור יחידה פונקציונלית עם כימוס מלא. הדוגמאות המובאות כאן וגם בחלקים הבאים עוזרות להבהיר עיקרון זה.

הערה

קטעי הקוד בפרק זה מציגים את העקרונות של תכנות מוכוון-עצמים. יש לזכור שקטעי קוד רבים אמנם כתובים ב-C#, אך העקרונות עצמם ישימים לכל שפה המוגדרת כ-OOP, ואינם ייחודיים לשפה מסוימת. למטרות השוואה בפרק זה, אציג דוגמאות גם בשפת C, אשר אינה מוכוונת-עצמים.

נניח שאתה כותב יישום המחשב את השכר של העובדת היחידה בשם Amy בחברתך. כדי לשייך נתונים מסוימים לעובדת זו, תכתוב בשפת C קוד, אשר קרוב לוודאי יהיה דומה לקוד הבא:

```
struct EMPLOYEE
{
      char szFirstName[25];
      char szLastName[25];
      int iAge; double
      dPayRate;
};
```

כך תחשב את השכר של Amy, על ידי שימוש במבנה EMPLOYEE:

```
void main()
{
	double dTotalPay;

	struct EMPLOYEE* pEmp;
	pEmp = (struct EMPLOYEE*)malloc(sizeof(struct EMPLOYEE));

	if (pEmp)
	{
		pEmp->dPayRate = 100;

		strcpy(pEmp->szFirstName, "Amy");
		strcpy(pEmp->szLastName, "Anderson");
		pEmp->iAge = 28;

		dTotalPay = pEmp->dPayRate * 40;
		printf("Total Payment for %s %s is %0.2f",
			pEmp->szFirstName, pEmp->szLastName, dTotalPay);
	}
	free(pEmp);
}
```

בדוגמה זו, הקוד מבוסס על נתונים הכלולים במבנה, וקוד כלשהו חיצוני (למבנה) המשתמש במבנה זה. ובכן, מה הבעיה? עיקר הבעיה הוא בהפשטה: המשתמש במבנה EMPLOYEE חייב לדעת הרבה מאוד על אודות הנתונים הנחוצים לעובד. מדוע? נניח שבזמן מאוחר יותר תרצה לשנות את הדרך לחישוב השכר של Amy. לדוגמה, אולי תרצה לכלול מרכיבי מס בתחשיב השכר נטו. יהיה עליך לשנות את כל הקוד המשתמש במבנה EMPLOYEE, וכן יהיה עליך לתעד את עובדת השינוי, לידיעת מתכנתים אחרים.

כעת נתבונן על גרסת C# של דוגמה זו:

```
using System;

class Employee
{
	public Employee(string firstName, string lastName, int age, double payRate)
	{
		this.firstName = firstName;
		this.lastName = lastName;
		this.age = age;
		this.payRate = payRate;
	}
```

```
        protected string firstName;
        protected string lastName;
        protected int age;
        protected double payRate;

        public double CalculatePay(int hoursWorked)
        {
            // Calculate pay here.
            return (payRate * (double)hoursWorked);
        }
}

 class EmployeeApp
{
        public static void Main()
        {
            Employee emp = new Employee ("Amy", "Anderson", 28, 100);
            Console.WriteLine("\nAmy's pay is $" + emp.CalculatePay(40));
        }
}
```

בגרסת C# של תוכנית הדוגמה EmployeeApp, המשתמש באובייקט יכול לקרוא לשיטה CalculatePay כדי שהאובייקט יחשב את השכר שלו עצמו (של האובייקט). היתרון בגישה זו הוא שהמשתמש אינו צריך לדאוג לדרך החישוב הפנימית של השכר. אם תחליט בעתיד לשנות את דרך חישוב השכר, לא תהיה לכך שום השפעה על הקוד הקיים. רמה זו של הפשטה היא אחת היתרונות הבסיסיים של השימוש באובייקטים.

כמובן שניתן לפשט את קוד הלקוח ב-C על ידי יצירת פונקציה שתאפשר לגשת למבנה EMPLOYEE. עם זאת, העובדה שיש ליצור פונקציה זו בנפרד לגמרי מהמבנה, היא בדיוק הבעיה. בשפה מוכוונת-עצמים כמו C#, הנתונים של האובייקט והשיטות הפועלות על אותם נתונים (הממשק שלו) נמצאים תמיד יחד.

יש לזכור שרק שיטות של אובייקט צריכות לשנות את המשתנים שלו. כפי שניתן לראות בדוגמה הקודמת, כל משתנה מחלקה של Employee מוגדר עם מציין הגישה **protected** (מוגן), חוץ מהשיטה CalculatePay המוגדרת **public** (ציבורית). מצייני גישה משמשים לציון רמת הגישה שיש למחלקות נגזרות ולקוד לקוח אל **חבר מחלקה** (class member) נתון. חבר מחלקה מוגן הינו זמין למחלקות נגזרות, אך אינו זמין לקוד לקוח חיצוני. חבר מחלקה ציבורי זמין גם למחלקות נגזרות וגם לקוד לקוח. בפרק 5 נדון ביתר פירוט במצייני גישה, אך כעת חשוב לזכור שמצייני גישה מאפשרים להגן על חברי מחלקה (או איברי מחלקה) חשובים מפני שימוש לא נכון בהם.