# 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**

**שמות מסחריים**

שמות המוצרים והשירותים המוזכרים בספר הינם שמות מסחריים רשומים של החברות שלהם. הוצאת הוד-עמי ו- Microsoft Press עשו כמיטב יכולתן למסור מידע אודות השמות המסחריים המוזכרים בספר זה ולציין את שמות החברות, המוצרים והשירותים. שמות מסחריים רשומים ( registered trademarks) המוזכרים בספר צוינו בהתאמה.

**#C הינו מוצר רשום של חברת Microsoft.**

**הודעה**

**ספר זה מיועד לתת מידע אודות מוצרים שונים. נעשו מאמצים רבים לגרום לכך שהספר יהיה שלם ואמין ככל שניתן, אך אין משתמעת מכך כל אחריות שהיא.**

**המידע ניתן "כמות שהוא" ("as is"). הוצאת הוד-עמי** ו- Microsoft Press **אינן אחראיות כלפי יחיד או ארגון עבור כל אובדן או נזק אשר ייגרם, אם ייגרם, מהמידע שבספר זה, או מהתקליטור שמצורף לו.**

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

- **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**

Copyright 2001 by Microsoft Corporation.
Original English language edition Copyright © 2001 by Microsoft Corporation.
All rights published by arrangement with the original publisher, Microsoft Press, a division of Microsoft Corporation, Redmond, Washington, USA.

©
**כל הזכויות שמורות**

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

**אין להעתיק או לשדר בכל אמצעי שהוא ספר זה או קטעים ממנו בשום צורה ובשום אמצעי אלקטרוני או מכני, לרבות צילום והקלטה, אמצעי אחסון והפצת מידע, ללא אישור בכתב מאת ההוצאה, אלא לשם ציטוט קטעים קצרים בציון שם המקור.**

הודפס בישראל 2002

All Rights Reserved
**HOD-AMI Ltd.**
P.O.B. 6108, Herzliya
ISRAEL, 2002

ISBN 965-361-296-4 מסת"ב

# תוכן עניינים מקוצר


מבוא ..... 13
הקדמה ..... 17

**חלק 1 - הנחת היסודות ..... 21**

פרק 1: יסודות התכנות מוכוון-עצמים ..... 23
פרק 2: מהו .NET של Microsoft ..... 43
פרק 3: מתחילים ..... 55

**חלק 2 - יסודות המחלקה של C# ..... 75**

פרק 4: CTS - Common Type System ..... 77
פרק 5: מחלקות - Classes ..... 89
פרק 6: שיטות ..... 123
פרק 7: מאפיינים (Properties), מערכים (Arrays) וסדרנים (Indexers) .. 143
פרק 8: תכונות - Attributes ..... 163
פרק 9: ממשקים - Interfaces ..... 181

**חלק 3 - כתיבת קוד ..... 207**

פרק 10: ביטויים ואופרטורים ..... 209
פרק 11: בקרת זרימה ..... 235
פרק 12: טיפול בשגיאות באמצעות חריגים ..... 265
פרק 13: העמסת אופרטורים והמרות מותאמות - Operator Overloading and User-Defined Conversions ... 287
פרק 14: נציגים ומטפלים באירועים - Delegates and Event Handlers .... 301

**חלק 4 - נושאים מתקדמים ב-C# ..... 321**

פרק 15: תכנות עם ריבוי מטלות - Multithreaded Programming ..... 323
פרק 16: Reflection ..... 347
פרק 17: Unmanaged Code ..... 365
פרק 18: Assemblies ..... 389

~~נספח: התקליטור המצורף ..... 405~~

Index ..... 413

# תוכן עניינים


מבוא ..... 13

הקדמה ..... 17

**חלק 1 - הנחת היסודות ..... 21**

פרק 1: יסודות התכנות מוכוון-עצמים ..... 23

כל דבר הוא אובייקט ..... 25

אובייקטים לעומת מחלקות ..... 29

instantiation ..... 29

שלושת העיקרים של שפות תכנות מוכוונות-עצמים ..... 31

encapsulation ..... 31

Inheritance ..... 34

Polymorphism ..... 37

סיכום ..... 41

פרק 2: מהו NET. של Microsoft ..... 43

פלטפורמת NET. של Microsoft ..... 43

סביבת העבודה NET. ..... 44

Windows DNA ו-NET. ..... 44

Common Language Runtime ..... 45

ספריות המחלקה בסביבת העבודה NET. ..... 46

MSIL ו- JITers ..... 48

Common Type System ..... 50

Metadata ו-Reflection ..... 50

אבטחה ..... 51

Deployment ..... 52

Unmanaged Code ..... 52

סיכום ..... 53

פרק 3: מתחילים ..... 55
כתיבת יישום #C הראשון ..... 55
בחירת עורך ..... 55
"Hello, World" ..... 58
השימוש במהדר שורת הפקודה ..... 58
הפעלת היישום ..... 59
סקירת הקוד ..... 60
תכנות במקום אחד ..... 60
Classes ו- Members ..... 61
השיטה Main ..... 61
השיטה System.Console.WriteLine ..... 62
Namespaces ו- using Directive ..... 62
Skeleton code ..... 63
אי-בהירות של מחלקות ..... 64
שגיאות ופתרונן ..... 65
Compile-Time Errors ..... 65
ILDASM ..... 66
"Hello, World" ב-MSIL ..... 67
קווים מנחים לתכנות ב- #C ..... 70
מתי להגדיר מרחבי שמות משלך ..... 70
קווים מנחים לבחירת שמות ..... 70
תקני מוסכמות לקביעת שמות ..... 71
סיכום ..... 74

חלק 2 - יסודות המחלקה של #C ..... 75

פרק 4: CTS - Common Type System ..... 77
כל דבר הוא אובייקט ..... 77
Value Types ו-Reference Types ..... 78
Value Types ..... 78
Reference Types ..... 79
Boxing ו-Unboxing ..... 79
המקור לכל הסוגים : System.Object ..... 80
Types ו-Aliases ..... 81
Casting Between Types ..... 82
Namespaces ..... 85
מילת המפתח using ..... 85
יתרונות CTS ..... 86
תפעוליות שיתופית בין שפות ..... 86
היררכיה המבוססת על מחלקה בודדת ..... 87
Type Safety ..... 87
סיכום ..... 88

## פרק 5: מחלקות - Classes ..... 89

הגדרת מחלקות ..... 89
Class Members ..... 90
Access Modifiers ..... 91
השיטה Main ..... 92
ארגומנטים של שורת-פקודה ..... 93
ערכים מוחזרים ..... 94
הרבה מחלקות עם השיטה Main ..... 94
Constructors ..... 95
Static Members ו- Instance Members ..... 97
Constructor Initializers ..... 99
Constants vs. Read-Only Fields ..... 102
Constants ..... 102
Read-Only Fields ..... 103
Garbage Collector ..... 105
מעט היסטוריה ..... 106
Deterministic Finalization ..... 107
ביצועים ..... 108
הפתרון המושלם ..... 114
פתרון שהוא (כמעט) מושלם ..... 115
Dispose Design Pattern ..... 116
Inheritance ..... 117
Multiple Interfaces ..... 119
Sealed Classes ..... 120
סיכום ..... 121

## פרק 6: שיטות ..... 123

הפרמטרים ref ו-out ..... 123
Method Overloading ..... 128
שיטה עם מספר לא ידוע של פרמטרים ..... 131
Virtual Methods ..... 132
Method Overriding ..... 133
Polimorphism ..... 134
Static Method ..... 140
גישה לחברי מחלקה ..... 142
סיכום ..... 142

פרק 7: מאפיינים (Properties), מערכים (Arrays) וסדרנים (Indexers) ..... 143
Properties as Smart Fields ..... 143
הגדרת מאפיינים ושימוש בהם ..... 144
מה באמת עושה כאן המהדר ..... 146
מאפיינים לקריאה בלבד ..... 148
הורשת מאפיינים ..... 149
שימוש מתקדם במאפיינים ..... 149
מערכים ..... 150
הצהרה על מערכים ..... 150
דוגמה של מערך חד-מימדי ..... 151
מערכים רב-מימדיים ..... 152
אחזור מספר המימדים ..... 154
מערכים משוננים ..... 155
Indexers ..... 157
הגדרת סדרנים ..... 158
דוגמה לסדרן ..... 158
קווים מנחים לתכנון ..... 160
סיכום ..... 161
פרק 8: תכונות - Attributes ..... 163
הצגת תכונות ..... 164
הגדרת תכונות ..... 165
קריאת מידע של תכונות ..... 166
תכונות של מחלקות ..... 166
תכונות של שיטות ..... 169
תכונות של שדות ..... 171
פרמטרים של תכונות ..... 173
פרמטרים לפי מיקום ופרמטרים לפי שם ..... 173
שגיאות נפוצות עם פרמטרים לפי שם ..... 175
סוגים חוקיים של פרמטרים עבור תכונות ..... 175
התכונה AttributeUsage ..... 176
הגדרת יעד התכונה ..... 176
תכונות לשימוש יחיד ולשימוש מרובה ..... 178
הגדרת חוקי הורשה לתכונה ..... 179
Attribute Identifiers ..... 179
סיכום ..... 180

פרק 9: ממשקים - Interfaces ..... 181
השימוש בממשק ..... 182
הצהרה על ממשקים ..... 183
מימוש ממשקים ..... 184
אחזור מידע אודות יישום ממשקים באמצעות is ..... 186
אחזור מידע אודות מימוש ממשקים באמצעות as ..... 190
שיוך מפורש של שם חבר ממשק ..... 193
הסתרת שמות באמצעות ממשקים ..... 193
מניעת אי-בהירות עם שמות ..... 196
ממשקים והורשה ..... 200
איחוד ממשקים ..... 203
סיכום ..... 205

חלק 3 - כתיבת קוד ..... 207

פרק 10: ביטויים ואופרטורים ..... 209
הגדרת אופרטורים ..... 209
קדימות אופרטורים ..... 210
קביעת קדימות ב- C# ..... 211
אסוציאטיביות ימנית ושמאלית ..... 211
שימוש מעשי ..... 212
אופרטורים של C# ..... 213
אופרטורים בסיסיים של ביטויים ..... 213
אופרטורים מתמטיים ..... 218
אופרטורים של יחסים ..... 226
אופרטורים פשוטים של הצבה ..... 229
סיכום ..... 233

פרק 11: בקרת זרימה ..... 235
משפטי בחירה ..... 235
משפט if ..... 235
ריבוי תנאים במשפט if ..... 237
אכיפת חוקי if ב- C# ..... 238
משפט switch ..... 240
צירוף תוויות case ..... 242
מניעת גלישה במשפטי switch ..... 244
משפטי איטרציה ..... 245
משפט while ..... 245
משפט do/while ..... 247
משפט for ..... 249
משפט foreach ..... 252

הסתעפות עם משפטי דילוג ..... 254
משפט break ..... 254
יציאה מלולאות אינסופיות ..... 255
משפט continue ..... 257
משפט goto ..... 258
סיכום ..... 264

## פרק 12: טיפול בשגיאות באמצעות חריגים ..... 265

סקירה כללית על טיפול בחריגים ..... 265
תחביר בסיסי של טיפול בחריגים ..... 267
Throwing an Exception ..... 267
Catching an Exception ..... 267
Rethrowing an Exception ..... 268
ניקוי בעזרת finally ..... 270
השוואה בין טכניקות לטיפול בשגיאות ..... 270
היתרון של טיפול בחריגים על פני קודים מוחזרים ..... 271
טיפול בשגיאות בהקשר הנכון ..... 273
שיפור קריאות הקוד ..... 274
Throwing Exceptions From Constructors ..... 276
השימוש במחלקה System.Exception ..... 276
בניית אובייקט Exception ..... 276
השימוש במאפיין StackTrace ..... 279
איך לתפוס סוגי חריגים מרובים ..... 280
גזירת מחלקות Exception מותאמות ..... 281
תכנון קוד עם טיפול בחריגים ..... 283
סוגיות תכנון של בלוק try ..... 283
סוגיות תכנון של בלוק catch ..... 285
סיכום ..... 286

## פרק 13: העמסת אופרטורים והמרות מותאמות - Operator Overloading and User-Defined Conversions ..... 287

Operator Overloading ..... 287
תחביר ודוגמה ..... 288
האופרטורים שניתן להעמיס ..... 291
הגבלות על העמסת אופרטורים ..... 291
קווים מנחים לתכנון ..... 292
User-Defined Conversions ..... 292
תחביר ודוגמה ..... 293
סיכום ..... 299

## פרק 14: נציגים ומטפלים באירועים - Delegates and Event Handlers ..... 301

שימוש בנציגים כשיטות משוב ..... 301
הגדרת נציגים כחברים סטטיים ..... 305
יצירת נציגים על פי צורך ..... 307
הרכבת נציגים ..... 309
הגדרת אירועים באמצעות נציגים ..... 315
סיכום ..... 319

# חלק 4 - נושאים מתקדמים ב-C# ..... 321

## פרק 15: תכנות עם ריבוי מטלות - Multithreaded Programming ..... 323

יסודות השימוש במטלות ..... 323
מטלות וריבוי משימות ..... 324
context switching ..... 324
יישום C# עם ריבוי מטלות ..... 325
עבודה עם מטלות ..... 326
AppDomain ..... 326
המחלקה Thread ..... 327
תזמון מטלות ..... 330
בטיחות וסינכרון מטלות ..... 334
הגנה על קוד באמצעות המחלקה Monitor ..... 335
משפט lock ..... 339
סינכרון קוד באמצעות המחלקה Mutex ..... 341
בטיחות מטלות ומחלקות NET ..... 343
קווים מנחים לשימוש במטלות ..... 343
מתי להשתמש במטלות ..... 343
מתי אין להשתמש במטלות ..... 344
סיכום ..... 345

## פרק 16: Reflection ..... 347

היררכיית API של השיקוף ..... 347
המחלקה Type ..... 348
קבלת אובייקט Type משם משתנה ..... 348
קבלת אובייקט Type משם של סוג ..... 349
Interrogating Types ..... 349
Assemblies ו-Modules ..... 352
ביצוע איטרציה על הסוגים של מכלל קוד ..... 352
פירוט המודולים של מכלל קוד ..... 355

Late Binding with Reflection ..... 357
יצירה וביצוע קוד בזמן ריצה ..... 360
סיכום ..... 363

## פרק 17: Unmanaged Code ..... 365

שירותי הפעלת פלטפורמות ..... 366
הצהרה על פונקציית DLL חיצונית ..... 366
Callbacks ו- PInvoke ..... 369
Marshalling ו- PInvoke ..... 370
Unsafe Code ..... 371
Pointers ..... 372
המשפט fixed ..... 373
COM ..... 375
עולם חדש ונועז ..... 375
מתחילים ..... 376
Metadata ו-Com typelib ..... 377
Early Binding to COM Components ..... 380
Query Interface ..... 382
Late Binding to COM Components ..... 383
COM Threading Models ..... 385
סיכום ..... 386

## פרק 18: Assemblies ..... 389

סקירה כללית ..... 389
Manifest Data ..... 390
Assembly Packing ..... 391
Assembly Deployment ..... 391
Assembly Versioning ..... 392
Building Assemblies ..... 392
Multiple Modules ..... 393
Creating Shared Assemblies ..... 395
Global Assembly Cache ..... 397
עיון במטמון ..... 397
Versioning Assemblies ..... 399
QFEs and the Default Version Policy ..... 402
Safe Mode Configuration File ..... 402
סיכום ..... 404

## ~~נספח: התקליטור המצורף ..... 405~~

## Index ..... 413

# מבוא

## #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, Visual 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- Microsoft C# בשם Microsoft 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"מ, 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

# פרק 1

# יסודות התכנות מוכוון-עצמים

פרק זה יצעיד אותך אל עולם המושגים של **תכנות מוכוון-עצמים** – **OOP** (Object-Oriented Programming), או **מוכוון 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 מוכוון-עצמים מושתת עליהם. #C היא אחת מהשפות האלו: היא תוכננה מיסודה להיות שפה מוכוונת-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 מוכוון-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 מוכוונות-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 מוכוון-עצמים הוקרבו בשפת ++C לטובת תמיכתה בקוד הקיים שנכתב בשפת C. אפילו לשפת Java, טובה ככל שתהיה, יש מספר מגבלות כשפה מוכוונת-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.

## אובייקטים לעומת מחלקות

ההבדל בין מחלקה לבין אובייקט מהווה מקור לבלבול למי שעוסק לראשונה בתכנות מוכוון-עצמים. כדי להמחיש את ההבדל בין שני מונחים אלה, נשנה את הדוגמה EmployeeApp כך שתהיה מציאותית יותר, אם נניח שהחברה עשויה להעסיק יותר מעובד אחד; כלומר, נתייחס לחברה המעסיקה קבוצת עובדים.

באמצעות שפת C ניתן היה להגדיר מערך של עובדים – על בסיס המבנה EMPLOYEE. מכיון שמספר המועסקים בחברה אינו קבוע, היינו יכולים ליצור מערך כזה עם מספר קבוע של אלמנטים, לדוגמה 10,000. אולם, מכיון שבחברה מועסקת כרגע עובדת אחת (Amy) בלבד, אין זו הדרך לניצול היעיל ביותר של משאבים. במקום זאת, הדרך הרגילה היתה ליצור רשימה מקושרת של מבנים מסוג EMPLOYEE ולהקצות זיכרון באופן דינמי, על פי הנדרש ביישום המשכורות.

הנקודה היא, שאנו עושים בדיוק את מה שאיננו אמורים לעשות. אנו מבזבזים אנרגיית תכנון במחשבות על השפה ועל המחשב שבו תפעל – במונחים של כמות הזיכרון הדרוש, ומתי – במקום להתמקד במרחב הבעיה. על ידי שימוש באובייקטים, אנו יכולים להתמקד ב-business logic ולא בצדדים הטכניים של פתרון הבעיה.

יש דרכים רבות להגדיר מחלקה ולהבדילה מאובייקט. ניתן לחשוב על מחלקה כעל סוג משתנה (בדומה ל-char, int או long) עם שיטות משלו. אובייקט הוא **מופע** (instance) של סוג משתנה או של מחלקה. עם זאת, ההגדרה החביבה עלי ביותר היא שמחלקה היא תסריט עבור אובייקט. מפתח התוכנה יוצר תסריט, כפי שמהנדס יוצר תסריט לפני תכנון בית. מרגע שהתסריט הושלם, יש רק תסריט אחד לכל טיפוס של בית, אך מספר לא מוגבל של אנשים יכולים לרכוש את התסריט ולבנות אותו טיפוס של בית. בדרך דומה, מחלקה היא תסריט לקבוצה נתונה של פונקציונליות, וכל אובייקט הנוצר על בסיס מחלקה מסוימת מקבל את כל הפונקציונליות של אותה מחלקה.

## instantiation

**יצירת מופע** (instantiation) היא מונח ייחודי לתכנות מוכוון-עצמים ופירושו יצירת מופע של מחלקה. המופע הוא אובייקט. בדוגמה הבאה ניצור מחלקה, או מיפרט לאובייקט. במילים אחרות, בשלב זה לא נעשית הקצאת זיכרון, כי אנו עוסקים בתסריט עצמו ולא באובייקט הממשי.

```
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);
    }
}
```

כדי ליצור מופע של המחלקה ולהשתמש בו, עלינו להצהיר על המופע בשיטה דומה לזו:

```
public static void Main()
{
    Employee emp = new Employee ("Amy", "Anderson", 28, 100);
}
```

בדוגמה זו, emp מוגדר כמשתנה מסוג Employee ויוצרים מופע שלו באמצעות האופרטור **new**. המשתנה emp מייצג מופע של המחלקה Employee ו**נחשב לאובייקט** של Employee. לאחר יצירת המופע ניתן לתקשר עם האובייקט באמצעות חברי המחלקה הציבוריים שלו. לדוגמה, נוכל לקרוא לשיטה CalculatePay של האובייקט emp. לא נוכל לעשות כך אם אין לנו אובייקט ממשי (יוצאים מן הכלל הם חברי מחלקה סטטיים, בהם נדון בפרקים 5 ו- 6).

נתבונן בקוד #C הבא:

```
public static void Main()
}
    Employee emp = new Employee();
    Employee emp2 = new Employee();
```

כאן יצרנו שני מופעים – emp ו- emp2 – של אותה המחלקה Employee. אף על פי שלכל אובייקט יש אותן היכולות מבחינת תכנותית, כל מופע של מחלקה יכיל מופע נתונים משלו ויטופל בנפרד. בדרך דומה, נוכל ליצור מערך שלם או אוסף של אובייקטים כאלה מסוג Employee. על מערכים תוכל ללמוד בפירוט בפרק 7. עם זאת, הנקודה החשובה לציון היא שרוב השפות מוכוונות-העצמים תומכות ביכולת להגדיר מערך של אובייקטים.

הודות לכך ניתן לקבץ אובייקטים בקלות ולבצע עליהם איטרציה על ידי קריאה לשיטות של מערך האובייקטים, או על ידי שימוש באינדקסים. השווה זאת לעבודה הרבה הדרושה לבניית רשימה מקושרת, שבה יש צורך לקשר ידנית כל פריט ברשימה לפריט שבא לפניו או אחריו.

# שלושת העיקרים של שפות תכנות מוכוונות-עצמים

על פי ההגדרה של Bjarne Stroustrup, יוצר שפת התכנות ++C, כדי ששפה תיקרא מוכוונת-עצמים היא חייבת לתמוך בשלושה עקרונות: אובייקטים, מחלקות והורשה. עם זאת, במרוצת הזמן הוגדרו שפות מוכוונות-עצמים כשפות הבנויות על המשולש של **כימוס** (encapsulation), **הורשה** (inheritance) **ורב-צורתיות** (polymorphism). הסיבה לשינוי זה נעוצה בכך שבמהלך השנים הובהר שכימוס ורב-צורתיות הם מרכיבים עיקריים ומהותיים בבניית מערכות מוכוונות-עצמים באותה מידה כמו מחלקות והורשה.

## encapsulation

כפי שהזכרתי קודם, **כימוס** (encapsulation), הנקרא לעתים **הסתרת מידע**, הוא היכולת להסתיר את כל הפנימיות של אובייקט ממשתמשיו ולספק ממשק רק לאותם חברים (members), של האובייקט או איברים שאנו מעוניינים שהמשתמש הלקוח יוכל לשנות באופן ישיר. עם זאת, מכיון שכבר הצגתי את ההפשטה באותו ההקשר, בחלק זה אסיר את הבלבול בקשר לשני עקרונות דומים אלה. כימוס מספק את הגבול שבין הממשק החיצוני של מחלקה – כלומר, החברים הציבוריים הגלויים למשתמשי המחלקה – לבין פרטי המימוש הפנימי. היתרון בכימוס עבור מפתח המחלקה הוא ביכולת לחשוף את אותם חברי המחלקה שיישארו סטטיים, או ללא שינוי, מחד, ולהסתיר את האיברים הפנימיים הדינמיים והמשתנים של המחלקה מאידך. כפי שראית כבר, הכימוס ב-#C מתבצע בעזרת השמה של מצייני גישה – public (ציבורי), private (פרטי), או protected (מוגן) – לכל חבר מחלקה (class member).

### Abstraction

**הפשטה** (abstraction) מתייחסת לאופן שבו מיוצגת בעיה נתונה במרחב התוכנית. שפות תכנות לכשעצמן מספקות הפשטה. מתי לאחרונה היית מוטרד אודות האוגרים והמחסניות של המעבד? גם אם בתחילה למדת לכתוב תוכניות באסמבלר, יש סיכוי גבוה מאוד שעבר זמן רב מאז שהיית צריך לעסוק בפרטי המחשב והחומרה. הסיבה לכך היא שרוב שפות התכנות משחררות אותך מן הצורך לעסוק ברמה הנמוכה, ובכך מאפשרות להתמקד במרחב הבעיה.

שפות מוכוונות-עצמים מאפשרות להכריז על מחלקות עם שמות וממשקים אשר מחקים בקירוב רב את יישויות מרחב הבעיה בעולם האמיתי, כך שלשימוש באובייקטים יש יותר נגיעה "טבעית" אליהם. התוצאה של הסרת האלמנטים שאין להם קשר ישיר לפתרון הבעיה הוא מתן האפשרות להתמקד באופן מוגדר יותר בבעיה והגברת היעילות. למעשה, בניסוח אחר לדבריו של Bruce Eckel בספר Thinking in Java, היכולת לפתור את רוב הבעיות תתכנס בדרך כלל לאיכות של ההפשטה שבה משתמשים.

עם זאת, לפנינו רק רמה אחת של הפשטה. כמפתח מחלקות עליך לחשוב גם במונחים של תכנון הפשטה עבור משתמשי המחלקה שלך בדרך הטובה ביותר, על מנת לאפשר ללקוחות להתמקד במטרה ולא לעסוק בפרטים הנוגעים לדרך הפעולה של המחלקה עצמה. בנקודה זו מן הראוי לשאול, "איך ממשק מחלקה מתייחס להפשטה?" – ממשק המחלקה הוא המימוש של ההפשטה.

כדי לסייע בגיבוש מושגים אלה אשתמש באנלוגיה מוכרת במידת מה מקורסי תכנות: הפעולה הפנימית של מכונות מכירה אוטומטיות. כדי לבצע את המוטל עליה, המכונה צריכה לקבל סכום כסף מזומן ולאשר אותו, להחזיר עודף ולמסור את המוצר שהלקוח בחר. אולם, למכונת מכירה יש קבוצת פונקציות סופית שעליה להציג בפני המשתמשים. ממשק זה מתבטא בחריץ למטבעות, לחצנים לבחירת הפריט הרצוי, מנוף לבקשת עודף, תא עבור העודף המוחזר ותא למסירת הפריט הנבחר. כל אחד מאלה מייצג חלק בממשק המכונה. מכונות מכירה אוטומטיות לא השתנו רבות מאז המצאתן. זאת משום שלמרות העובדה שהמערכת הפנימית שלהם השתנתה עם התפתחות הטכנולוגיה, הרי שהממשק הבסיסי שלהן לא היה צריך להשתנות במידה רבה. חלק אינטגרלי של תכנון ממשק מחלקה הוא הבנה עמוקה של מרחב הבעיה. הבנה זו תעזור ליצור ממשק המספק למשתמשים גישה למידע ולשיטות שהם זקוקים להן, אך מבודד אותם מהעבודה הפנימית של המחלקה. עליך לתכנן ממשק לא רק כדי לפתור את בעיות ההווה, אלא גם לבודד את פנימיות המחלקה במידה מספקת, כך שניתן יהיה לשנות חברי מחלקה פרטיים ללא הגבלה בלי שתהיה לכך השפעה על קוד קיים.

היבט נוסף, חשוב באותה מידה, בתכנון הפשטה למחלקה הוא התחשבות מתמדת במתכנת יישום הלקוח. נניח שאתה כותב מנוע כללי של מסד נתונים. אם אתה "גורו" של מסדי נתונים, קרוב לודאי שתרגיש נוח עם מילים כמו cursors, commitment control ו- tuples. עם זאת, רוב המפתחים שאינם מנוסים בתחום, לא יהפכו למבינים במונחים אלה. על ידי שימוש במונחים זרים ללקוחות המחלקה שלך, אתה מסכל את כל תכלית ההפשטה – להגדיל את היצרנות של המתכנת על ידי הצגת מרחב הבעיה במונחים טבעיים.

דוגמה אחרת לזמן שבו צריך לחשוב על הלקוח היא כאשר מחליטים מי מחברי המחלקה יהיו זמינים עבורו. ושוב, עם מעט ידע על מרחב הבעיה ועל לקוחות המחלקה שלך ההחלטות נעשות ברורות. בדוגמת מנוע מסד הנתונים הקודמת, קרוב לוודאי שלא תרצה לאפשר ללקוחותיך גישה ישירה לחברי מחלקה המייצגים חוצצי נתונים פנימיים (internal data buffers). האופן שבו מוגדרים חוצצי נתונים אלה עשוי להשתנות בקלות בעתיד. בנוסף לכך, מכיון שחוצצים אלה קריטיים לפעולה הכוללת של המנוע שלך, תרצה להבטיח שהם יעודכנו באמצעות השיטות שלך בלבד. בדרך זו תוכל להיות בטוח שננקטו כל אמצעי הזהירות הדרושים.

נוטים לעיתים לחשוב שמערכות מוכוונות-עצמים מתוכננות קודם כל כדי להקל על יצירת מחלקות. למרות שתכונה זו אכן מעלה את היצרנות בטווח הקצר, הרווח בטווח הארוך מגיע רק לאחר ההכרה ש-OOP דרוש כדי להקל על עבודת התכנות של לקוחות המחלקה. בתכנון המחלקות התחשב תמיד במתכנת שעומד לממש אותן (ליצור מופעים שלהן) או לגזור מהן.

## יתרונות

עיצוב ההפשטה של מחלקות בדרך המועילה ביותר למתכנתים המשתמשים בהן הוא בעל חשיבות עליונה בעיצוב תוכנה הניתנת לשימוש חוזר. ממשק יציב, סטטי ועקבי לאורך שינויי יישום עתידיים של המחלקה מחייב פחות שינויים עם הזמן. במקרה של האובייקט Employee והפונקציונליות של תלוש המשכורת בדוגמת הקוד הקודמת, רק שיטות מעטות יהיו רלוונטיות כמו CalculatePay (חישוב שכר), GetAddress (קבלת כתובת העובד) ו- GetEmployeeType (קבלת סוג העובד). אם אתה מבין את מרחב הבעיה של יישום משכורות, תוכל לקבוע בקלות וברמה גבוהה למדי את השיטות שיהיו נחוצות למשתמשי המחלקה. אם בתכנון המחלקה אתה משלב ידע מעמיק על מרחב הבעיה ביחד עם ראיית הנולד ותכנון קדימה, מובטח לך במידה סבירה שחלק גדול מהממשק שיצרת לא ישתנה למרות שינויים עתידיים ביישום הממשי של המחלקה. אחרי הכל, מזווית הראייה של המשתמש, זו רק המחלקה Employee, והיתרון מבחינתו הוא שכמעט כלום לא צריך להשתנות מגירסה לגירסה.

ההפרדה בין המשתמש לבין פרטי היישום היא זו שהופכת מערכת שלמה ומורכבת לקלה יותר להבנה ולכן גם לאחזקה. השווה זאת לשפות פרוצדורליות כמו C, שבה כל מודול צריך לציין את השמות ולגשת בצורה מפורשת לחברים של מבנה נתון. במקרה זה, בכל פעם שמשנים את חברי המבנה, חייבים לשנות גם כל שורה בקוד הפונה לאותו מבנה.

# Inheritance

**הורשה** (inheritance) מתייחסת ליכולת של המתכנת לציין שלמחלקה אחת יש **סוג של קירבה** למחלקה אחרת. באמצעות הורשה ניתן ליצור (או לגזור) מחלקה חדשה המבוססת על מחלקה קיימת. אחר כך ניתן לשנות את המחלקה הנגזרת בדרך הרצויה וליצור אובייקטים חדשים מסוגה. יכולת זו היא התמצית של יצירת היררכיית מחלקות. מלבד הפשטה, הורשה היא החלק החשוב ביותר בתכנון מקיף של מערכת. **מחלקה נגזרת** (derived class) היא המחלקה החדשה שנוצרה, **ומחלקת הבסיס** (base class) היא המחלקה שממנה נגזרה המחלקה החדשה. המחלקה הנגזרת החדשה יורשת את כל החברים של מחלקת הבסיס, וכך מאפשרת ניצול חוזר של עבודה קודמת.

הערה

ב-#C, הזמינות של חברי מחלקת הבסיס למחלקות נגזרות, נשלטת על ידי מצייני הגישה המשמשים בהגדרת החבר (member). סוגיה זו תידון בפירוט בפרק 5. למטרות הדיון שלפנינו, נניח שהמחלקה הנגזרת יורשת את כל חברי מחלקת הבסיס שלה.

כדי ללמוד מתי ואיך להשתמש בהורשה, נתייחס שוב לדוגמה הקודמת EmployeeApp. קרוב לוודאי שיש סוגים שונים של עובדים, כמו עובדים בשכר חודשי, עובדים לפי שעות ועובדים לפי חוזה. למרות שכולם אובייקטים מסוג Employee ובעלי ממשק זהה, ברוב המקרים הם יתפקדו בתוכם באופן שונה זה מזה. לדוגמה, השיטה CalculatePay תפעל עבור עובדים בשכר חודשי באופן שונה מאשר עבור עובדים לפי חוזה. עם זאת, אתה מעוניין באותו ממשק CalculatePay עבור המשתמשים שלך, ללא התחשבות מראש בסוג העובד.

אם רק התחלת בתכנות מוכוון-עצמים, אתה עשוי להעלות בדעתך פתרון פשוט לכאורה לבעיה, עם מבנה נתונים הכולל חבר המציין את סוג העובד. כך לדוגמה היה נראה קוד הפונקציה:

```
Double CalculatePay(EMPLOYEE* pEmployee, int iHoursWorked)
{
   // Validate pEmployee pointer.

   if (pEmployee->type == SALARIED)
   {
      // Do W-2 employee processing.
   }
   else if (pEmployee->type == CONTRACTOR)
   {
      // Do 1099 processing.
   }
   else if (pEmployee-> == HOURLY)
```

```
    {
        // Do hourly processing.
    }
    else
    {
        // Do corp-to-corp processing.
    }

    // Return the value from one of the
    // compound statements above.
}
```

בקוד זה יש מספר בעיות. ראשית, הצלחת הביצוע של הפונקציה CalculatePay קשורה באופן הדוק למבנה EMPLOYEE. כפי שציינתי כבר, הצמדה הדוקה כזו מהווה בעיה, כי כל שינוי במבנה EMPLOYEE גורם לשבירת הקוד. כמתכנת בגישה מוכוונת-עצמים, הדבר האחרון שתרצה לעשות הוא להכביד על משתמשי המחלקה שלך עם הצורך לדעת את פרטי התכנון המסובכים שלה. הדבר דומה ליצרן מכונת מכירה אוטומטית המחייב אותך להבין את המכניקה הפנימית של המכונה, כדי שתוכל לקנות באמצעותה משקה סודה.

שנית, הקוד אינו מאפשר שימוש חוזר. מרגע שמתחילים להבין כיצד הורשה מעודדת שימוש חוזר בקוד, מתברר שאובייקטים ומחלקות הם רכיבי תכנות טובים. במקרה זה, מגדירים במחלקת הבסיס את כל החברים המתפקדים באופן זהה ללא תלות בסוג העובד. כל מחלקה שנגזרת ממחלקת הבסיס יורשת את הפונקציונליות הזו, ומשנה רק את מה שנחוץ. כך זה נראה בקוד ב-C#:

```
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 SalariedEmployee : Employee
{
    public string SocialSecurityNumber;

    public void CalculatePay (int hoursWorked)
    {
        // Calculate pay for a W-2 employee.
    }
}

class ContractEmployee : Employee
{
    public string FederalTaxId;

    public void CalculatePay (int hoursWorked)
    {
        // Calculate pay for a contract employee.
    }
}
```

שלוש תכונות בדוגמה זו ראויות לציון:

- מחלקת הבסיס Employee מגדירה חבר בשם EmployeeId, אשר יורשות אותו המחלקות הנגזרות SalariedEmployee ו-ContractEmployee. מחלקות אלו אינן עושות דבר בכדי לקבל את EmployeeId – הן יורשות אותו באופן אוטומטי כתוצר לוואי של גזירתן מהמחלקה Employee.
- שתי המחלקות הנגזרות מיישמות גרסאות CalculatePay משלהן. עם זאת, שים לב ששתיהן יורשות את הממשק, ולמרות שהן משנות את החלק הפנימי בכדי להתאימו לצרכים הייחודיים של כל אחת מהן, קוד המשתמש נשאר ללא שינוי.
- שתי המחלקות הנגזרות מוסיפות חברים לחברים שכבר התקבלו בירושה ממחלקת הבסיס. המחלקה SalariedEmployee מגדירה את המחרוזת SocialSecurityNumber, והמחלקה ContractEmployee מוסיפה את הגדרת החבר FederalTaxId.

זו היתה דוגמה פשוטה לכך שהורשה מאפשרת שימוש חוזר בקוד, באמצעות הורשת פונקציונליות ממחלקת הבסיס. בנוסף לכך, באפשרותך להרחיב את המחלקה על ידי הוספת משתנים ושיטות משלך.

## הגדרה נכונה של הורשה

בסעיף זה אתייחס לעיקרון החשוב של הורשה נכונה על ידי שימוש במונח של Marshall Cline ו-Greg Lomow מתוך הספר C++ FAQs: **יכולת המרה** (substitutability). פירושו של מונח זה הוא, שההתנהגות הידועה של המחלקה הנגזרת יכולה להמיר את זו של מחלקת הבסיס. חשוב לרגע על משפט זה – זהו הכלל החשוב ביותר והיחיד שתלמד על בניית היררכיית מחלקות אשר עומדת במבחן הזמן, ומקיימת את ההבטחות של OOP בכל הנוגע לקוד הניתן לשימוש חוזר ולהרחבה.

כלל נוסף שיש לזכור כאשר יוצרים היררכיות של מחלקות: **מחלקה נגזרת דורשת לא יותר ומבטיחה לא פחות ממחלקת הבסיס שלה בכל ממשק שעובר בהורשה**. אי ציות לכלל זה גורם לשבירת הקוד הקיים. ממשק של מחלקה הוא חוזה מחייב בינו לבין משתמשי המחלקה. כאשר למתכנת יש מצביע למחלקה נגזרת, הוא תמיד יכול להתייחס אל מחלקה זו כאילו היא מחלקת הבסיס. הדבר נקרא **המרה כלפי מעלה** (upcasting). בדוגמה שלנו, אם ללקוח יש מצביע לאובייקט ContractEmployee, אזי יש לו גם מצביע משתמע לבסיס של אותו אובייקט, לאובייקט Employee. לכן על פי ההגדרה, ContractEmployee צריך תמיד להיות מסוגל לתפקד כמו מחלקת הבסיס שלו. שים לב שכלל זה מתייחס רק לפונקציונליות של מחלקת הבסיס. מחלקה נגזרת יכולה להוסיף התנהגות מגבילה יותר ביחס לדרישות של עצמה ולהבטיח מעט ככל שתחפוץ. לפיכך, הכלל חל רק על חברים שעוברים בהורשה, כי לקוד הקיים יהיה חוזה רק עם חברים אלה.

# Polymorphism

ההגדרה התמציתית ביותר ששמעתי עבור **רב-צורתיות** (polymorphism) היא "פונקציונליות המאפשרת לקוד ישן לקרוא לקוד חדש". זו אולי התועלת הגדולה ביותר של תכנות מוכוון-עצמים, כי כך אפשר להרחיב או להגדיל את המערכת מבלי לשנות או לפגוע בקוד קיים.

נניח שאתה כותב שיטה (method) שצריכה לבצע איטרציה על אוסף אובייקטים מסוג Employee ולקרוא לשיטה CalculatePay של כל אחד מהם. אין כאן כל בעיה, כל זמן שבארגון יש רק סוג אחד של עובדים, מכיון שניתן להכניס רק את האובייקט המסוים לתוך האוסף (collection). אולם, מה קורה כאשר מתחילים להעסיק עובדים מסוגים אחרים? לדוגמה, אם המחלקה Employee היא מימוש של הפונקציונליות עבור עובד בשכר חודשי, מה תעשה כשתתחיל להעסיק עובדים לפי חוזה אשר את משכורותיהם יש לחשב באופן שונה? ובכן, בשפה פרוצדורלית עליך לשנות את הפונקציה, כך שתטפל גם בסוג העובד החדש, כי הקוד הישן קרוב לוודאי אינו יודע לטפל בקוד החדש. פתרון מוכוון-עצמים מטפל בהבדלים אלה באמצעות רב-צורתיות.

אם נחזור לדוגמה שלנו, עליך להגדיר מחלקת בסיס בשם Employee. לאחר מכן עליך להגדיר מחלקה נגזרת עבור כל סוג של עובד (כפי שראינו קודם). כל מחלקה שנגזרת מ-Employee תיישם את השיטה CalculatePay על פי צרכיה, וכאן מתרחש הקסם. עם רב-צורתיות, אם יש מצביע משתמע לאובייקט ואתה קורא לשיטה של אותו אובייקט, מערכת זמן הריצה של השפה תבטיח קריאה לגירסה הנכונה של השיטה. הנה הקוד הממחיש את הדברים:

```
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 virtual double CalculatePay(int hoursWorked)
   {
      Console.WriteLine("Employee.CalculatePay");
      return 42; // bogus value
   }
}

class SalariedEmployee : Employee
{
   public SalariedEmployee(string firstName, string lastName,
                           int age, double payRate)
   : base(firstName, lastName, age, payRate)
   {}

   public override double CalculatePay(int hoursWorked)
   {
      Console.WriteLine("SalariedEmployee.CalculatePay");
      return 42; // bogus value
   }
}
```

```
class ContractorEmployee : Employee
{
   public ContractorEmployee(string firstName, string lastName,
                             int age, double payRate)
   : base(firstName, lastName, age, payRate)
   {}

   public override double CalculatePay(int hoursWorked)
   {
      Console.WriteLine("ContractorEmployee.CalculatePay");
      return 42; // bogus value
   }
}

class HourlyEmployee : Employee
{
   public HourlyEmployee(string firstName, string lastName,
                         int age, double payRate)
   : base(firstName, lastName, age, payRate)
   {}
   public override double CalculatePay(int hoursWorked)
   {
      Console.WriteLine("HourlyEmployee.CalculatePay");
      return 42; // bogus value
   }
}
class PolyApp
{
   protected Employee[] employees;

   protected void LoadEmployees()
   {
      Console.WriteLine("Loading employees...");

      // In a real application, we'd probably get this
      // from a database.
      employees = new Employee[3];

      employees[0] = new SalariedEmployee ("Amy", "Anderson", 28, 100);
      employees[1] = new ContractorEmployee ("John", "Maffei", 35, 110);
      employees[2] = new HourlyEmployee ("Lani", "Ota", 2000, 5);
```

```
        Console.WriteLine("\n");
    }

    protected void CalculatePay()
    {
        foreach(Employee emp in employees)
        {
            emp.CalculatePay(40);
        }
    }

    public static void Main()
    {
        PolyApp app = new PolyApp();

        app.LoadEmployees();
        app.CalculatePay();
    }
}
```

הידור (compilation) והפעלת היישום מניבים את הפלט הבא:

```
c:\>PolyApp

Loading employees...

SalariedEmployee.CalculatePay
ContractorEmployee.CalculatePay
HourlyEmployee.CalculatePay
```

שים לב שרב-צורתיות מעניקה שני יתרונות לפחות. ראשית, היא נותנת לך את היכולת לקבץ אובייקטים שיש להם מחלקת בסיס משותפת ולהתייחס אליהם בצורה עקבית. בדוגמה הקודמת, למרות שמבחינה טכנית לפנינו שלושה סוגים שונים של אובייקטים – ContractEmployee, ContractEmployee ו-HourlyEmployee – ניתן להתייחס אל כולם כאל אובייקטים של Employee, כי כולם נגזרים ממחלקת הבסיס Employee. כך אפשר לקבץ את כולם למערך המוגדר כמערך של אובייקטים מסוג Employee. בגלל הרב-צורתיות, כאשר קוראים לשיטה של אחד מהאובייקטים האלה, מערכת זמן הריצה תבטיח קריאה לשיטה הנכונה של האובייקט הנגזר.

את היתרון השני כבר ציינתי בתחילת סעיף זה: קוד ישן יכול להשתמש בקוד חדש. שים לב שהשיטה PolyApp.CalculatePay מבצעת איטרציה על מערך של אובייקטים מסוג Employee. מכיון ששיטה זו מחלצת את האובייקטים כאובייקטים משתמעים מסוג Employee ויישום זמן הריצה של רב-צורתיות מבטיח קריאה לשיטה הנכונה של המחלקה הנגזרת, ניתן להוסיף למערכת סוגי עובדים נגזרים אחרים, להכניס אותם למערך אובייקטים מסוג Employee, והקוד כולו ימשיך לפעול בלי שיהיה צורך לשנות את הקוד המקורי!

# סיכום

פרק זה לקח אותך לסיור דרך המונחים והעקרונות של תכנות מוכוון-עצמים. דיון מלא על הנושא היה מחייב מספר פרקים והיה מסיט אותך מהמיקוד בנושאי ספר זה. עם זאת, הבנה מוצקה של היסודות חיונית למיצוי מקסימלי של שפת #C.

בפרק זה סקרנו מספר מושגים. המפתח להבנת מערכות מוכוונות-עצמים נמצא בהבנת ההבדל בין מחלקות, אובייקטים וממשקים, ואיך מושגים אלה נקשרים לפתרונות יעילים. פתרונות מוכווני-עצמים טובים תלויים ביישום שלושת העקרונות של תכנות מוכוון-עצמים: כימוס, הורשה ורב-צורתיות. המושגים שהוצגו בפרק זה מניחים את המצע לפרק הבא, המציג את מסגרת NET. של Microsoft ואת סביבת זמן הריצה Common Language Runtime.

# פרק 2 מהו NET. של Microsoft

הבנה טובה של **NET.** (בטא זאת "dot-net". בהמשך נרשה לעצמנו לשם הפשטות לכתוב "NET" בלבד), והבנת חלקה של שפת #C ביוזמה זו של חברת Microsoft, תאפשר לך להבין באופן מלא כמה מהיסודות של #C הנתמכים על ידי זמן הריצה של NET. הסקירה שתוצג בפרק זה תעזור לך להבין גם את המושגים המופיעים במהלך הספר ומדוע תכונות מסוימות של שפת #C פועלות בדרך שהן פועלות.

אם אתה קורא קבוצות דיון או רשימות דיוור בנושא NET, תיווכח שקיים בלבול בקרב חלק מהמשתמשים בכל הנוגע למושגים הקשורים לטכנולוגיה זו. כשהשמות מעורפלים ולעיתים סותרים, יש קושי לעקוב אחר השחקנים. ללא ספק, חלק מהבעיה נובע מהעובדה שהנושא חדש. על כן, אסביר תחילה כמה מושגים הקשורים בטכנולוגיית NET.

## פלטפורמת NET. של Microsoft

הרעיון העומד מאחורי NET. של Microsoft הוא הסטת מוקד התכנות מעולם שבו התקנים יחידים ואתרי Web מחוברים זה לזה באופן פשוט באמצעות האינטרנט, אל עולם שבו התקנים, שירותים ומחשבים פועלים יחד בכדי לספק פתרונות מלאים עבור משתמשים. הפתרון של NET מורכב מארבעה מרכיבי יסוד:

- שירותי אבני בניין של NET, או גישה תכנותית לשירותים מסוימים, כמו אחסון קבצים, לוח פגישות ו- Passport.NET (שירות לאימות זהות).
- תוכנת התקנים של NET, אשר תפעל בהתקני אינטרנט חדשים.
- חוויית המשתמש של NET, אשר כוללת תכונות כמו הממשק הטבעי, סוכני מידע, וטכנולוגיה של תגיות חכמות, המבצעת אוטומציה של היפר-קישורים למידע המתייחס למילים ומשפטים במסמכים הנוצרים על ידי המשתמש.

- תשתית NET (.NET infrastructure) המורכבת מסביבת העבודה NET (.NET Framework), Microsoft Visual Studio.NET, NET Enterprise Servers ו- Microsoft Windows.NET.

תשתית NET היא החלק שאליו מתייחסים רוב המפתחים בהקשר של NET. אתה יכול להניח שבכל זמן שאני מתייחס ל-NET (ללא שם תואר מקדים) אני מתכוון לתשתית NET. תשתית זו מתייחסת לכל הטכנולוגיות המרכיבות את סביבת העבודה החדשה ליצירה ולהפעלה של יישומים בעלי עוצמה ומבוזרים, הניתנים להרחבה. החלק של NET המאפשר לנו לפתח יישומים אלה היא סביבת העבודה NET.

סביבת העבודה NET מורכבת מסביבת זמן הריצה **CLR** (Common Language Runtime) ומספריות המחלקה של מסגרת NET, הנקראות לעיתים Base Class Library (BCL). ניתן לחשוב על CLR כעל המכונה הווירטואלית שבה מתפקדים יישומי NET. ספריות המחלקה של סביבת העבודה NET זמינות לשימושן של כל שפות NET. אם אתה מכיר את Microsoft Foundation Class (MFC) או את Object Windows Library (OWL) של Borland, אזי המושג של ספריות מחלקה כבר ידוע לך. ספריות המחלקה של סביבת העבודה NET כוללות תמיכה בכל דבר, החל מקלט/פלט (I/O) של קבצים ומסדי נתונים ועד ל-XML ו-SOAP. למעשה, ספריות אלו הן כה נרחבות, עד כי דרוש ספר שלם רק כדי לתת סקירה שטחית על כל המחלקות הנתמכות.

כהערת ביניים, כשאני משתמש במונח "מכונה וירטואלית" (virtual machine), איני מתכוון למכונה הווירטואלית של Java (JVM). למעשה, אני משתמש בהגדרה המסורתית של המונח. לפני כמה עשורים, עוד לפני ש-Java היתה יותר ממילה למשקה שחור וחם, הטביעה חברת IBM לראשונה את המונח "מכונה וירטואלית". מכונה וירטואלית היתה הפשטה ברמה גבוהה של מערכת הפעלה, אשר בתוכה היו יכולות לתפקד מערכות הפעלה אחרות בסביבה מכומסת (encapsulated) לגמרי. בהתייחסותי ל-CLR כאל סוג של מכונה וירטואלית, אני מתכוון לעובדה שהקוד הפועל בתוך CLR פועל למעשה בסביבה מכומסת ומנוהלת, בנפרד מתהליכים אחרים במחשב.

# סביבת העבודה NET.

הבה נכיר את סביבת העבודה NET ומה היא נותנת לנו. תחילה אשווה את העבודה בסביבת NET לעבודה בסביבת פיתוח קודמת ליישומים מבוזרים. לאחר מכן אעבור על רשימת היכולות אשר NET מספקת למפתחי יישומים, כדי ליצור מהר יותר יישומים מבוזרים.

## Windows DNA ו-NET.

האם הניסוח שבו השתמשתי קודם לתיאור NET – "הסביבה החדשה ליצירת והפעלת יישומים בעלי עוצמה ומבוזרים הניתנים להרחבה" – נשמע מוכר? אם כן, יש סיבה לכך: ביסודו של דבר, NET היא הצאצא של ניסיון קודם לספק את היעדים הנשגבים האלה.

פלטפורמה זו נקראה Windows DNA. עם זאת, NET היא הרבה יותר ממה ש- Windows DNA התכוונה להיות. Windows DNA היתה פלטפורמה שהתמקדה בפתרון בעיות עסקיות באמצעות מוצרי השרת של Microsoft. לפעמים היו משתמשים במילה "דבק" בצירוף ל- Windows DNA, כמו במשפט: "DNA מגדיר את הדבק המשמש לאחות ביחד מערכות בעלות עוצמה ומבוזרות הניתנות להרחבה". עם זאת, חוץ מהיותה מפרט טכני, או ארכיטקטורה כללית, לא היו ב- Windows DNA כל מרכיבים ממשיים. זהו רק הבדל אחד ממספר הבדלים עיקריים בין Windows DNA לבין NET. NET של Microsoft אינה רק אוסף של מפרטים. היא גם כוללת מספר מוצרים מוחשיים כמו מהדרים (compilers), ספריות מחלקה, ואפילו אוסף יישומים למשתמש הקצה.

## Common Language Runtime

**CLR** (**C**ommon **L**anguage **R**untime) היא לב ליבה של NET. כפי שמציין השם, זו סביבת ריצה שבה יישומים שנכתבו בשפות שונות יכולים לפעול יחד ולהסתדר בצורה טובה. יכולת זו ידועה גם בשם **תפעוליות שיתופית בין שפות** (cross-language interoperability). CLR מספקת סביבה נוחה זו עבור יישומים משפות שונות בעזרת אוסף חוקים הנקרא **CLS** (**C**ommon **L**anguage **S**pecification). כל מהדר שפה חייב לציית לחוקים אלה כדי שיוכל ליצור יישומי NET הפועלים ב-CLR. כלומר, לפעול בצורה תקינה לצד כל יישום NET אחר ובאותה מידה של שיתופיות.

עיקרון חשוב המתייחס ל-CLR נקרא **קוד מנוהל** (managed code). קוד מנוהל הוא הקוד שפועל תחת חסות CLR ולכן הוא מנוהל על ידי CLR. חשוב על זה באופן הבא: בסביבות Microsoft Windows של ימינו פועלים תהליכים שונים לגמרי זה מזה. כל היישומים צריכים לציית לחוק אחד בלבד: להתנהג יפה בסביבת Windows. יישומים אלה נוצרים על ידי שימוש באחד מתוך מספר רב של מהדרים שונים לגמרי זה מזה. במילים אחרות, היישומים צריכים לציית רק לחוקים הכלליים ביותר, כדי שיוכלו לפעול תחת Windows.

לסביבת Windows יש מספר חוקים גלובליים הקשורים לאופן שבו היישומים צריכים להתנהג במונחים של תקשורת זה עם זה, הקצאת זיכרון, או אפילו גיוס מערכת ההפעלה של Windows לביצוע עבודה מועילה עבורם. עם זאת, בסביבת קוד מנוהל נקבעו מספר חוקים בכדי להבטיח שכל היישומים יתנהגו באופן אחיד ברמה הגלובלית, בלי להתחשב בשפות שבהן הם נכתבו. ההתנהגות האחידה של יישומי NET היא התמצית של NET למזלי ולמזלך, חוקים גלובליים אלה משפיעים בעיקר על כותבי המהדרים.

# ספריות המחלקה בסביבת העבודה NET.

ספריות המחלקה בסביבת העבודה NET חשובות עד מאד ליכולת השיתופיות בין יישומים, כי הן מאפשרות למפתחים להשתמש בממשק תכנות יחיד לכל הפונקציונליות ש-CLR חושפת. אם אי-פעם פיתחת תוכנות עבור Windows ביותר משפה אחת, קרוב לוודאי שתאהב תכונה זו. למעשה, ספריות המחלקה בסביבת העבודה NET מתוות מגמה מהפכנית בפיתוח מהדרים. לפני NET, רוב מפתחי המהדרים פיתחו שפה בעלת יכולת לבצע את רוב העבודה של עצמה. אפילו לשפה כמו ++C, אשר תוכננה כהקבצה של פונקציונליות לשימוש במשולב עם ספריית מחלקה, יש מידה מסוימת של פונקציונליות משלה. אולם, בעולם של NET, השפות הופכות להיות מעט יותר מאשר ממשקים תחביריים לספריות המחלקה של NET.

לשם דוגמה, נתבונן תחילה ביישום "Hello, World" כתוב בשפת ++C:

```
#include <iostream.h>

int main(int argc, char* argv[])
{
   cout << "Hello, World!" << endl;
   return 0;
}
```

הפונקציה main המהווה נקודת כניסה של כל יישום ++C/C, משתמשת בפונקציה cout כדי לכתוב "Hello, World" להתקן הפלט הסטנדרטי. לשם כך כולל היישום תחילה את קובץ הכותר (header file) שבו נמצאת ההצהרה על cout. עם זאת, הנקודה החשובה כאן היא שלא ניתן לכתוב יישום זה בשפות של NET ללא ספריות המחלקה של סביבת NET. אכן, לשפות NET אין אפילו את התכונות הבסיסיות ביותר של מהדר, כמו היכולת להציג מחרוזת טקסט על מסך. מבחינה טכנית, הפונקציה cout מיושמת בזמן הריצה של ++C/C, שהיא בעצמה מחלקה. אולם, משימות בסיסיות של ++C כמו עיצוב מחרוזות, קלט/פלט של קבצים וקלט/פלט של תצוגה, נחשבות, לפחות מבחינה לוגית, חלק משפת הבסיס. השפה #C – או כל שפת NET אחרת – כמעט ואינה יכולה לבצע אפילו עבודות פשוטות ללא ספריית המחלקה של סביבת NET.

בכדי להמחיש זאת, נתבונן בדוגמה "Hello, World" הכתובה ב-#C:

```
class Hello
{
   public static void Main()
   {
      System.Console.WriteLine("Hello, World");
   }
}
```

אם כך, מה המשמעות של אוסף ספריות מחלקה משותף זה, והאם הוא טוב עבורך? ובכן, הכל תלוי בנקודת המבט שלך. אוסף ספריות משותף פירושו שמבחינה תיאורטית, כל השפות הן בעלות יכולות זהות כי כולן חייבות להשתמש בספריות מחלקה אלו כדי לבצע כל משימה, חוץ מהצהרה על משתנים.

בקבוצת דיון הועלתה הטענה "מדוע צריך שפות רבות, אם יש לכולן את אותן היכולות?" איני מבין טענה זו. כמי שעבד בסביבות מרובות שפות, אני יכול להעיד על התועלת הגדולה בכך שלא צריך לזכור מה כל שפה עושה במערכת. בסופו של דבר, עבודתנו כמפתחים היא ליצור קוד, ולא לדאוג אם לשפה מועדפת יש יתרון כזה או אחר.

שאלה אחרת שאני שומע לעיתים קרובות היא "אם כל שפות NET עושות את אותו הדבר, מדוע אנו צריכים יותר מאחת?" התשובה קשורה לעובדה שמתכנתים הם יצורים בעלי הרגלים. מיקרוסופט בוודאי לא רצתה לבחור שפה אחת מתוך שפות זמינות רבות אחרות ולאלץ מיליוני מתכנתים להשליך הצידה את כל שנות ניסיונם בשפות אחרות. המתכנת נדרש לא רק ללמוד API חדש, אלא גם לשלוט בתחביר שונה לגמרי. במקום זה, המפתח יכול להמשיך להשתמש בשפה המתאימה ביותר לעבודתו. כאשר שם המשחק הוא יצרנות ויעילות, לשנות את מה שלא מחייב שינוי אינו חלק ממשוואה זו.

הערה

אף על פי שמבחינה תיאורטית ספריות המחלקה של סביבת NET מאפשרות למהדרים לגרום לכך שכל הפונקציונליות של ה-CLR (Common Language Runtime) תהיה זמינה למשתמשי השפה, אין זה תמיד כך. במיקרוסופט יש נקודת מחלוקת בין צוות תכנון ספריית המחלקות של סביבת NET והצוותים השונים העוסקים במהדרים והיא, שלמרות שצוות ספריות המחלקה ניסה לחשוף את כל הפונקציונליות שלהן לשפות שונות, צוותי המהדרים השונים נדרשו ליישם עמידה בסטנדרטים המינימליים של סביבת ההרצה בלבד. כאשר שאלתי מספר מפתחים אודות סתירה זו, נאמר לי שבמקום שלכל שפה תהיה גישה לכל הפונקציונליות של סביבת העבודה של NET, כל אחד מצוותי המהדרים החליט ליישם רק את התכונות שעל פי תחושתו הן הישימות ביותר עבור משתמשי אותו מהדר. למזלנו, נראה ש-C# היא השפה שמספקת ממשק כמעט לכל הפונקציונליות שיש לסביבת NET להציע.

# MSIL ו- JITers

כדי להקל על כותבי שפה לתרגם את שפותיהם ל- NET. פיתחה מיקרוסופט שפה דומה לשפת אסמבלי, אשר נקראת "שפת הביניים של מיקרוסופט" – **MSIL** (Microsoft Intermediate Language). כדי להדר יישומים ל-NET, המהדרים מקבלים קלט של קוד מקור ומייצרים פלט של MSIL, שהינה שפה בפני עצמה שניתן לכתוב בה יישומים. עם זאת, כמו עם אסמבלי, נראה שכמתכנת לא תעשה זאת לעולם חוץ מאשר בנסיבות מיוחדות. מכיון ש-MSIL היא שפה בפני עצמה, כל צוות מהדר מקבל החלטות משלו מה מידת התמיכה שהוא מאפשר עבורה. אולם, אם אתה כותב מהדר ורוצה ליצור שפה שאינה פועלת יחד עם שפות אחרות, עליך להגביל את עצמך רק לתכונות המוגדרות על ידי CLS (Common Language specifications).

כאשר מהדרים יישום #C או כל יישום הכתוב בשפה העומדת בחוקי CLS, היישום מהודר לקוד MSIL. קוד זה מתורגם אחר כך לפקודות CPU (קוד המכונה), כאשר היישום מופעל בפעם הראשונה על ידי ה-CLR (למעשה, רק הפונקציות הנקראות מתורגמות בפעם הראשונה שהן מופעלות). הבה נבדוק מה באמת קורה במערכת:

1. כותבים קוד מקור ב-#C.
2. מהדרים את הקוד לקובץ EXE באמצעות מהדר #C (csc.exe).
3. מהדר #C מייצר קוד MSIL ביחד עם מצהר (manifest) הכלול בחלק לקריאה בלבד של קובץ EXE חוקי, בעל כותר קובץ הפעלה נייד – PE (WIN32 Portable) סטנדרטי. עד כאן הכל בסדר. אולם, הנה החלק החשוב: כאשר המהדר מייצר את הפלט, הוא מייבא גם פונקציה בשם CorExeMain_ ממערכת זמן הריצה של NET.
4. כאשר היישום מופעל, מערכת ההפעלה טוענת את PE יחד עם ספריות קישור-דינמי (DLL) תלויות, כמו זו המייצאת את הפונקציה CorExeMain_ (mscoree.dll), בדיוק כפי שהיא עושה עם כל PE חוקי.
5. לאחר מכן, רכיב הטעינה של מערכת ההפעלה מדלג אל נקודת הכניסה בתוך PE, אשר הוצבה על ידי מהדר #C. ושוב, זו בדיוק הדרך שבה מבוצע כל PE אחר ב- Windows. עם זאת, מכיון שמערכת ההפעלה אינה יכולה לבצע קוד MSIL בוודאות, נקודת כניסה זו היא רק רכיב המפעיל את הפונקציה CorExeMain_ ב- mscoree.dll.
6. הפונקציה מתחילה את ביצוע קוד MSIL שהוכנס ל- PE.

7. מכיון שלא ניתן לבצע קוד MSIL ישירות – כי הוא אינו בפורמט קובץ ביצוע בקוד מכונה – CLR מהדרת את MSIL באמצעות מהדר מסוג JIT (just-in-time, בדיוק בזמן) לקוד טבעי של הוראות המעבד. הידור JIT מתרחש בזמן הקריאה לשיטות בתוכנית. קוד הביצוע המהודר מוטמן במחשב ומהודר שוב רק אם נעשה שינוי בקוד המקור.

ניתן להשתמש בשלושה סוגים שונים של מהדרי JIT (JITers) להמרת קוד MSIL לקוד מכונה, כתלות בנסיבות:

- **Install-time code generation** – **יצירת קוד בזמן התקנה** - יצירת קוד בזמן התקנה מהדרת מכלל קוד (assembly) בשלמותו לקוד בינארי התלוי במעבד, בדיוק כפי שעושה מהדר של C++. מכלל קוד הוא חבילת הקוד הנשלחת למהדר (בסעיף "deployment" נדון בפירוט בחבילות קוד אלו). הידור זה מתבצע בזמן ההתקנה, שבו המשתמש כמעט ואינו שם לב לתהליך ההידור של חבילת הקוד. היתרון ביצירת קוד בזמן התקנה הוא באפשרות להדר את חבילת הקוד במלואה רק פעם אחת לפני הפעלתה. מכיון שכל הקוד מהודר, אינך צריך לדאוג לנושא הביצועים המשתנים בכל פעם ששיטה בקוד שלך מתבצעת בפעם הראשונה. הדבר דומה לתוכנית נופש במתכונת שיתוף-זמן (time-share) שבה אתה משלם על כל דבר מראש. למרות שהתשלום עבור הנופש הוא מכאיב, אינך צריך לדאוג יותר לתשלום עבור הנופש. ההחלטה אם ומתי להשתמש במהדר זה תלויה בגודל המערכת המסוימת ובסביבת ההטמעה שלך. בדרך כלל, אם אתה עומד ליצור תוכנית התקנה למערכת שלך, עליך להשתמש במהדר זה, וכך המשתמש יקבל את הגירסה הטובה ביותר של המערכת.

- **JIT** - מהדר JIT הינו ברירת המחדל ומופעל בזמן ריצה בכל פעם ששיטה נקראת בפעם הראשונה, כפי שתיארתי קודם. הדבר דומה לתוכנית "שלם-מייד". זהו מהדר ברירת המחדל אם אינך מריץ באופן מפורש את מהדר PreJIT.

- **EconoJIT** - זהו סוג אחר של מהדר JIT לזמן ריצה, שתוכנן במיוחד עבור מערכות עם משאבים מוגבלים, כמו מחשבי כף יד שיש להם משאבי זיכרון מוגבלים. ההבדל העיקרי בין מהדר JIT זה לבין המהדר הרגיל הוא בשילוב אפשרות הנקראת code pitching, המאפשרת ל-EconoJIT להתעלם מקוד טבעי שנוצר אם הזיכרון הולך ואוזל. היתרון הוא בפינוי זיכרון, אך החיסרון הוא שאם קוד זה מופעל שוב, יש להדר אותו מחדש כאילו לא הופעל מעולם.

## Common Type System

אחת מתכונות המפתח של כל סביבת פיתוח היא מערכת סוגי הנתונים שלה. סביבת פיתוח שכוללת מספר סופי של סוגי נתונים, או מערכת המגבילה את יכולת המתכנת להרחיב את הסוגים שהיא מספקת, אינה בעלת תוחלת חיים ארוכה. סביבת זמן הריצה של NET נותנת יותר מאשר מערכת סוגים יחידה ואחידה בלבד, הניתנת לשימוש בכל השפות התואמות CLS. היא גם מאפשרת למפתחי שפות להרחיב את מערכת הסוגים על ידי הוספת סוגים חדשים הנראים ופועלים בדיוק כמו הסוגים המובנים במערכת. המשמעות עבורך כמפתח היא, שאתה יכול להשתמש בכל סוגי הנתונים באופן אחיד, מבלי להתחשב אם הם סוגי NET מוגדרים מראש או סוגים שנוצרו על ידי משתמש. (בפרק 4 אסביר את מערכת הסוגים ואיך המהדר של C# תומך בה).

## Reflection-ו Metadata

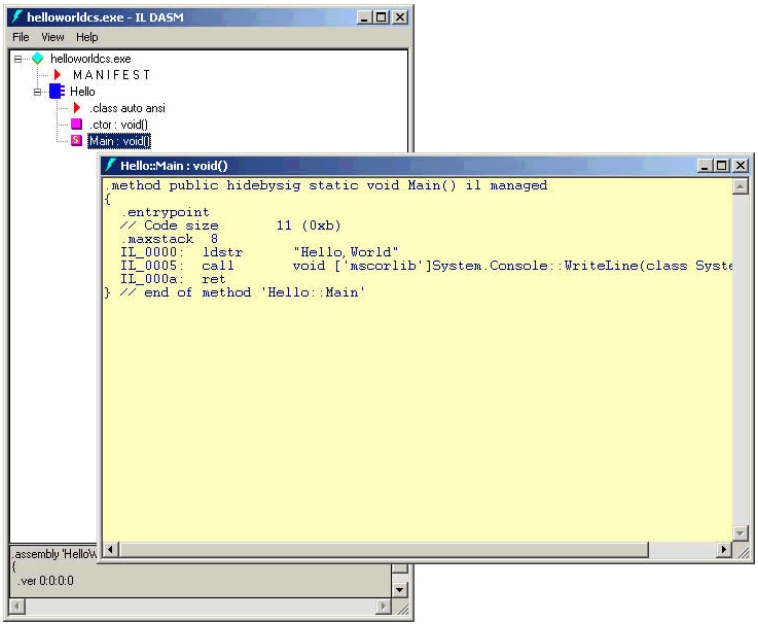
בסעיף "MSIL ו- JITers" הזכרתי שהמהדרים תואמי-CLS מקבלים קלט של קוד מקור ומייצרים קוד MSIL להידור (באמצעות JITers) וביצוע בזמן הריצה. בנוסף למיפוי קוד מקור לסדרות של הוראות MSIL, יש למהדרים תואמי-CLS משימה חשובה באותה מידה: הטבעת מטא-נתונים בתוך ה-EXE המתקבל.

**מטא-נתונים** (metadata) הם נתונים המתארים נתונים. בהקשר שלנו, מטא-נתונים הם אוסף הפריטים התכנותיים המהווים את EXE, כמו סוגי הנתונים המוצהרים והשיטות המיושמות. מטא-נתונים אלה דומים לספריות הסוג (typelibs) הנוצרות עם רכיבי COM. עם זאת, המטא-נתונים הנוצרים על ידי מהדר NET מספקים יותר מידע ומושלמים יותר באופן ממשי מספריות typelibs שאנו מורגלים בהם, והם גם תמיד מאוחסנים בתוך ה-EXE. בדרך זו, אין סיכוי לאיבוד המטא-נתונים של היישום או לקבלת צמדי קבצים לא תואמים.

הסיבה לשימוש במטא-נתונים היא פשוטה. הם מאפשרים למערכת זמן הריצה של NET לדעת בזמן הריצה איזה סוגי נתונים יוקצו ואילו שיטות ייקראו. הדבר מאפשר למערכת זמן הריצה להתקין את סביבת העבודה להפעלה יעילה יותר של היישום. האמצעי לקריאת מטא-נתונים אלה נקרא **שיקוף** (reflection). בפועל, ספריות המחלקה של מסגרת NET מספקות אוסף שלם של שיטות שיקוף המאפשרות לכל יישום – לא רק CLR – לקרוא מטא-נתונים של יישום אחר.

כלים כמו Visual Studio.NET משתמשים בשיטות שיקוף אלו ליישום תכונות, כמו IntelliSense. עם תכונה זו, כאשר אתה מקליד שם של שיטה, מוקפצת על המסך תיבת רשימה עם כל הארגומנטים שלה. Visual Studio.NET מרחיק לכת עם פונקציונליות זו ומציג גם את כל החברים של סוג הנתונים. בפרק 15 נעסוק ב-API של שיקוף.

כלי אחר שימושי מאד של NET, המנצל את יתרון השיקוף הוא IL Disassembler של סביבת NET (ILDASM). כלי עזר רב-עוצמה זה מנתח את המטא-נתונים של יישום היעד ומציג על המסך מידע על היישום בהיררכיה הדומה לעץ. תרשים 2-1 מציג איך נראה ב- ILDASM היישום "Hello, World" שנכתב ב-#C.




**תרשים 2-1** כך מציג ILDASM את היישום "Hello, World" שנכתב ב- #C

החלון המוצג ברקע בתרשים 2-1 הוא החלון הראשי של IL Disassembler. לחיצה כפולה על השיטה Main בתצוגת העץ פותחת את החלון שבקדמה, המציג את פרטי השיטה Main.

## אבטחה

הצד החשוב ביותר בכל סביבת פיתוח ליישומים מבוזרים הוא הטיפול באבטחה, וגם כאן יש ל-NET מה לחדש. האבטחה מופעלת מרגע שמחלקה נטענת על ידי CLR, כי רכיב ההטענה של המחלקה הוא חלק מתוכנית האבטחה של NET. לדוגמה, כאשר מחלקה נטענת בזמן ריצה של NET, מאומתים גורמים הקשורים לאבטחה כמו חוקי נגישות ודרישות של עקביות עצמית. בנוסף, בדיקות אבטחה מבטיחות שלקטע קוד יש אישור מתאים לגשת למשאבים מסוימים. קוד האבטחה מבטיח הגדרת תפקיד ומידע זיהוי. בדיקות אבטחה אלו נמשכות מעבר לגבולות התהליך והמעבד, בכדי להבטיח שלא תהיה כל פשרה בנוגע לנתונים רגישים בסביבות מחשוב מבוזרות.

## Deployment

**הטמעה** (deployment) היא משימה קשה בהקשר של פיתוח מערכות מבוזרות גדולות ורחבות מאד. כל מפתח בסביבת Windows יכול לספר, שהטיפול בקבצים בינאריים שונים, נושאי רישום, רכיבי COM, והתקנת ספריות תמיכה במוצרים – כמו קישוריות פתוחה של מסדי נתונים (ODBC) ואובייקטי גישה לנתונים (DAO) – מספיקים בהחלט בכדי לגרום תעסוקה ללא גבול. על כן, צוות התכנון של NET הקדיש זמן רב מאד לתחום ההטמעה.

המפתח להטמעת יישום NET הוא המושג של **מכללי קוד** (assemblies). מכללי קוד הם חבילות של פונקציונליות קשורה מבחינה סמנטית, הנבנות כיישויות של קובץ יחיד או קבצים מרובים. אופן ההטמעה המדויק של יישום תלוי בייעודו, אם מפתחים יישום של שרת Web או יישום שולחן עבודה רגיל ל-Windows. עם זאת, בעת הצגת מכלל הקוד כאוסף פונקציונליות מכומס במלואו, הטמעה יכולה להיות פשוטה עד כדי העתקת מכללי הקוד הנחוצים למיקום היעד.

בעיות רבות שגרמו לטרחה מרובה עבור מתכנתים לפני שפותחה סביבת העבודה NET סולקו כעת. לדוגמה, אין צורך ברישום רכיבים – כפי שעושים עם רכיבי COM ופקדי Microsoft ActiveX – כי כל הרכיבים מתארים את עצמם הודות למטא-נתונים ושיקופים. מערכת זמן הריצה של NET גם עוקבת אחר הקבצים וגרסאות הקבצים המשויכים ליישום. לכן, כל יישום מותקן משויך באופן אוטומטי לקבצים שהם חלק ממכלל הקוד שלו. אם תוכנית התקנה מנסה לכתוב על קובץ הדרוש ליישום אחר, מערכת זמן הריצה של NET תאפשר לתוכנית ההתקנה להתקין את הקובץ הדרוש. אבל, CLR אינה מוחקת את גרסת הקובץ הקודמת, כי היא עדיין נדרשת ליישום הראשון.

## Unmanaged Code

**קוד לא מנוהל** (unmanaged code) אינו נשלט על ידי מערכת זמן הריצה של NET. לשם הבהרה, קוד זה עדיין פועל בזמן הריצה של NET. עם זאת, לקוד לא מנוהל אין את היתרונות שיש לקוד מנוהל, כמו אוסף פסולת (ניהול זיכרון אוטומטי), מערכת סוגים אחידה ומטא-נתונים. למרות חוסר ההיגיון שבכך, הרצת קוד לא מנוהל בסביבת NET נעשית מחוסר ברירה, כאשר עומדים בפני נסיבות שאינן מאפשרות הרבה.

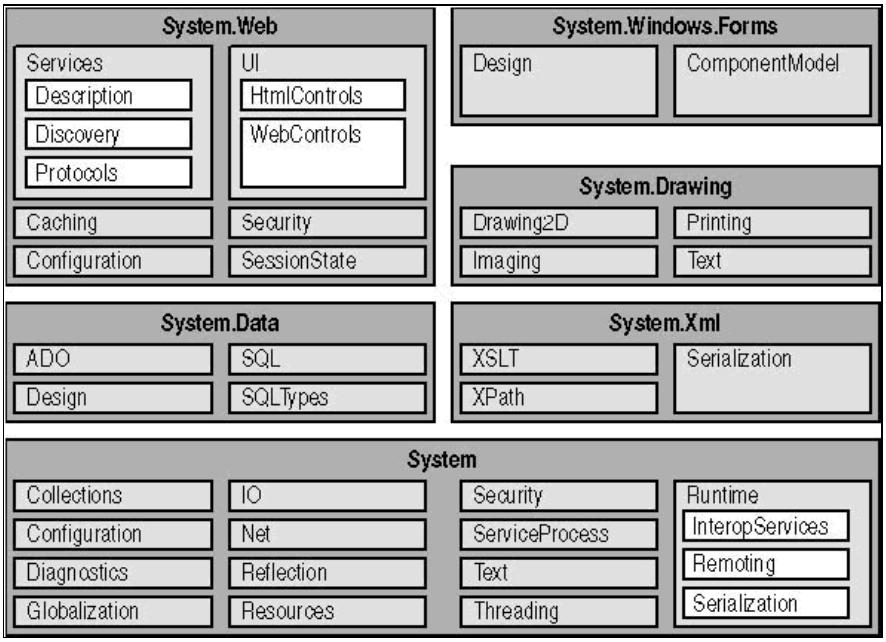
הנה מספר מצבים שבגינם יש להודות למיקרוסופט על הכנסת תכונה זו ל-NET:

- **קוד מנוהל הקורא לפונקציות DLL לא מנוהלות** – נניח שיישום צריך להיות עם ממשק ל-DLL דמוי-C, והחברה שכתבה DLL זה אינה מאמצת את NET באותה המהירות שחברתך עושה. במקרה זה, עדיין עליך לקרוא ל- DLL זה מתוך יישום NET. דוגמה זו בדיוק נסקרת בפרק 16.

- **קוד מנוהל העובד עם רכיבי COM** – מאותה הסיבה שאולי צריך את המשך התמיכה ביכולת לקרוא לפונקציות של DLL דמויי-C מיישומי NET, צריך להמשיך לתמוך גם ברכיבי COM. עושים זאת על ידי יצירת עטיפת NET לרכיבי COM, כך שהלקוח המנוהל חושב שהוא פועל עם מחלקה של NET. גם נושא זה נסקר בפרק 16.
- **קוד לא מנוהל המשתמש בשירותי NET** – זו בדיוק הבעיה ההפוכה – רוצים לגשת ל-NET מקוד לא מנוהל. הבעיה נפתרת באמצעות גישה הפוכה: מערימים על לקוח COM החושב שהוא משתמש בשרת COM, אך למעשה מדובר בשרת NET מסוג כלשהו. תוכל למצוא דוגמאות לכך בפרק 16.

# סיכום

תפיסת NET. של מיקרוסופט מציגה מעבר למודל תכנות שבו התקנים, שירותים ומחשבים פועלים יחד כדי לספק פתרונות למשתמשים. לב השינוי הוא הפיתוח של סביבת העבודה NET וסביבת זמן הריצה CLR, המוצגים בתרשים 2-2. סביבת NET מכילה את ספריות המחלקה, השיתופיות לכל השפות שעברו הידור, כדי לאפשר להן לפעול ב-CLR. מכיון ששפת #C תוכננה עבור CLR, אינך יכול לבצע ולו את המשימה הפשוטה ביותר ב-#C ללא ה-CLR וספריות המחלקה של סביבת NET. הבנת התכונות של טכנולוגיות אלו חיונית להפקת המירב מ-#C ומשאר פרקי ספר זה.




**תרשים 2-2** סביבת העבודה NET מכילה ספריות שתוכננו לאפשר תפעוליות שיתופית קלה יותר בין שירותים, התקנים ומחשבים

# פרק 3 מתחילים

פרק זה מהווה הקדמה ללב הנושא, הנידון בחלקי הספר הבאים. כאן תכיר את תהליך הפיתוח של יישום #C פשוט. תחילה אסקור את היתרונות והחסרונות של עורכי טקסט שתוכל להשתמש בהם כדי לכתוב תוכנית מקור ב-#C. מרגע שתבחר עורך, תכתוב את יישום הדוגמה הידוע "Hello, World" כדי להכיר את התחביר והמבנה המהווים בסיס לכתיבת יישומי #C. תוכל לראות שתחביר זה הוא בעל נוסח קבוע כמו ברוב השפות, וניתן להשתמש בו כתבנית לכתיבת רוב היישומים הבסיסיים בשפת #C. לאחר מכן תלמד להדר את הקוד באמצעות מהדר שורת הפקודה ולהפעיל את היישום החדש שלך.

## כתיבת יישום #C הראשון

הבה נעבור על השלבים ליצירה והפעלה של יישום #C הראשון.

### בחירת עורך

לפני שכותבים יישום ב-#C יש לבחור עורך טקסט. הסעיפים הבאים מתארים את העורכים השכיחים ביותר ומספקים מספר עובדות רלוונטיות אודות בחירת עורך לפיתוח ב-#C.

#### Notepad

Notepad של Microsoft היה עורך הטקסט המקובל ביותר במהלך השלבים הראשונים של #C בקרב מפתחים המשתמשים ב-SDK של סביבת העבודה NET לכתיבת יישומי #C.

השתמשתי ב-Notepad בספר זה מסיבות אחרות שאותן אפרט מאוחר יותר. עם זאת, איני ממליץ על השימוש ב-Notepad מסיבות אלו:

- קבצי #C יש לשמור עם הסיומת cs. בתיבת הדו-שיח File Save, אם אתה מנסה לשמור את הקובץ בשם כמו Test.cs ואינך מספיק זהיר, תקבל קובץ בשם test.cs.txt, אלא אם תזכור לשנות את הגדרת סוג הקובץ ברשימה Save As Type לסוג "All Files", או שתקיף את שם הקובץ במרכאות כפולות - "Test.cs".
- Notepad אינו מציג מספרי שורות והדבר מכביד בעת ניפוי שגיאות, כאשר המהדר מדווח על שגיאה בשורה מסוימת.
- Notepad מכניס 8 מרווחים עבור Tab, ולכן כתיבת כל דבר חוץ מ- "Hello, World" יכולה להקשות על קריאת יישומים.
- Notepad אינו מבצע הזחה אוטומטית של שורות כאשר אתה לוחץ על Enter. לכן, עליך להקיש על Tab עד לעמודה הרצויה, כדי להקליד את הקוד.

קיימות סיבות נוספות, אך כפי שאתה רואה, Notepad **אינו בחירה טובה** עבור פיתוח יישומי #C.

## Visual Studio 6

העורך Visual Studio כולל את כל התכונות הנדרשות לעריכה ולשמירה של קבצי #C. אחד מהיתרונות הגדולים בעורך המיועד למפתח תוכנה הוא הדגשה (הארה) של תחביר. עם זאת, מכיון שהעורך Visual Studio 6 שוחרר לשוק מספר שנים לפני #C ותוכנן לפיתוח יישומי Visual C++, יהיה עליך לבצע כוונון קצר של Visual Studio 6 כדי שידע להדגיש קוד #C בצורה נכונה. השלב הראשון הוא לשנות את מפתח הרישום (Registry Key) של Visual Studio. אתר את המפתח הבא ברישום שלך על ידי שימוש ב-Regedit.exe או עורך רישום אחר:

```
HKEY_CURRENT_USER\Software\Microsoft\DevStudio\6.0\
Text Editor\Tabs/Language Settings\C/C++\FileExtensions
```

המפתח יכלול מחרוזת בדומה לזו:

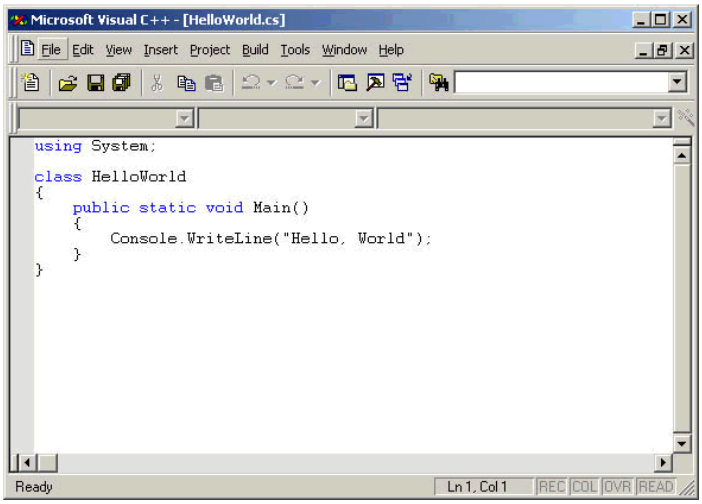
```
cpp;cxx;c;h;hxx;hpp;inl;tlh;tli;rc;rc2
```

הוסף את הסיומת cs בסוף הערך (שים לב שהתו נקודה-פסיק בסוף הערך הוא אופציונלי). ערך הרישום החדש יכול להיות בדומה לזה:

```
cpp;cxx;c;h;hxx;hpp;inl;tlh;tli;rc;rc2;cs
```

כעת, אם תפתח קובץ עם סיומת cs ב- Visual Studio 6, Visual Studio תזהה אותו כקובץ נתמך.

לאחר מכן עליך לציין ל- Visual Studio אילו מחרוזות הן מילות מפתח ב-C#. לשם כך אתה יוצר קובץ בשם usertype.dat ומציב אותו בספריה שבה נמצא msdev.exe. זהו קובץ טקסט מסוג ASCII המכיל את כל מילות המפתח שצריכות להיות מודגשות – מילת מפתח אחת בכל שורה. כאשר Visual Studio מופעל הוא טוען קובץ זה. לכן, כאשר אתה משנה את הקובץ, עליך להפעיל מחדש את Visual Studio כדי לראות את השינויים. תרשים 3-1 מציג איך ייראה קוד #C לאחר ביצוע שלבים אלה.




**תרשים 3-1** יתרון השימוש בעורך המספק הדגשת תחביר כמו ב- Visual Studio הוא משוב מיידי על הקלדות שגויות של מילות מפתח

## Visual Studio.NET

מובן מאליו שאם אתה רוצה יצרנות מקסימלית בסביבת העבודה NET, עליך להשתמש ב- Visual Studio.NET. מלבד הכלים האינטגרליים והאשפים ליצירת יישומי #C, עורך זה כולל גם כלים, כגון IntelliSense ו- Dynamic Help. עם IntelliSense, כאשר אתה מקליד שם של מרחב שמות (namespace) או שם מחלקה, חברי המחלקה מוצגים באופן אוטומטי, כך שאינך צריך לזכור כל חבר של כל מחלקה. IntelliSense גם מציג את כל הארגומנטים ואת סוגיהם כאשר אתה מקליד שם של שיטה ופותח סוגריים. אמנם, גם Visual Studio 6 מספק יכולת זו, אך הוא אינו תומך בוודאות במחלקות ובסוגי הנתונים של NET. Dynamic Help היא תכונה חדשה ב- Visual Studio. כאשר אתה מקליד קוד בעורך, חלון נפרד מציג נושאים הקשורים למילה שהסמן נמצא עליה. לדוגמה, אם אתה מקליד את מילת המפתח namespace, החלון מציג קישורים לנושאי עזרה הסוקרים את השימוש במילת המפתח namespace.

### עורכי טקסט אחרים

קיימים עורכים נוספים שפותחו על ידי בתי תוכנה שונים, כמו CodeWright של Starbase ו- Visual SlickEdit של MicroEdge. לא אכנס לדיון בעורכים אלה, אך תוכל להשתמש בכל אחד מהם לכתיבת יישומי #C.

### במה השתמשתי בספר זה

מכיון שהתחלתי בכתיבת הספר בזמן ש- Visual Studio.NET היתה בראשית דרכה, השתמשתי ב- Visual Studio 6. אני מקווה שהשימוש בסביבה שלא נוצרה עבור #C עזרה לי לקיים את הבטחתי שספר זה יהיה שימושי לכל מי שמפתח יישומי #C, ללא תלות בסביבת הפיתוח שהוא בוחר בה. כפי שציינתי קודם, אפשר להשתמש גם ב-Notepad כדי לכתוב את כל דוגמאות הספר ולקבל את התוצאות המצופות.

## "Hello, World"

בהנחה שבחרת את סביבת הפיתוח שלך, הבה נתבונן ביישום #C הראשון שתכתוב: "Hello, World". כתוב את הקוד הבא ושמור אותו בקובץ בשם HelloWorld.cs:

```
class HelloWorld
{
   public static void Main()
   {
      System.Console.WriteLine("Hello, World");
   }
}
```

תחילה נתרכז בהצגת היישום הראשון, הידורו וביצועו. אם תסיים שלבים אלה בהצלחה, תוכל להיות סמוך ובטוח שאתה נמצא בסביבה הנכונה ליצירה והפעלה של יישומי #C. לאחר מכן נעסוק בפרטי כל שורה בקוד.

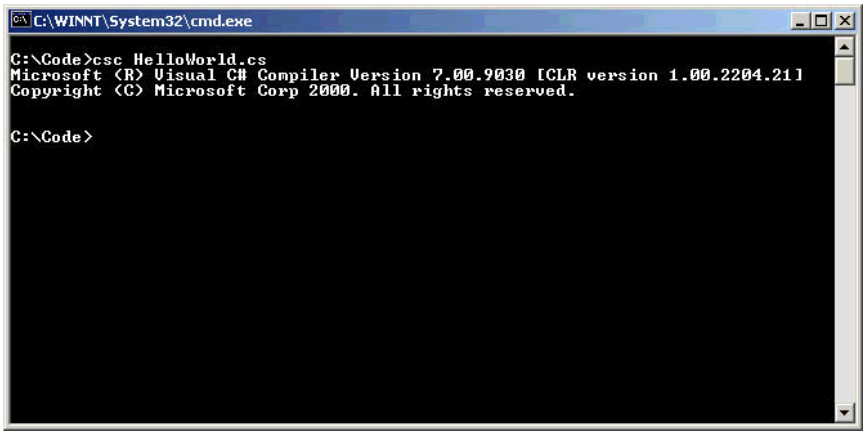
## השימוש במהדר שורת הפקודה

אם אתה משתמש בעורך בעל תכונה מובנית לבניית יישומי #C, שלב זה עשוי להיות מיותר. עם זאת, לאורך כל הספר אשתמש במהדר שורת הפקודה של #C (csc.exe), למרות שבעבודתך בוודאי תשתמש בעורך מתקדם יותר. יש לכך שני יתרונות. ראשית, אין זה משנה באיזו סביבה אתה משתמש, השלבים לבניית הדוגמאות יפעלו תמיד. שנית, לימוד האפשרויות השונות של המהדר יעזור לך בטווח הארוך במצבים שבהם העורך שלך אינו מספק לך שליטה מלאה על שלב הפעילות.

לאחר הקלדת הקוד של "Hello, World", פתח את חלון הפקודה (cmd.exe) ונווט אל הספריה שבה נמצא הקובץ HelloWorld.cs. כעת הקלד את השורה הזו:

```
csc HelloWorld.cs
```

אם הכל פועל כשורה, תראה תוצאות דומות לאלו המוצגות בתרשים 3-2, שבהן שם וגרסת המהדר מוצגים ביחד עם הודעות שגיאה או אזהרות כלשהן. ללא חשיבות בגרסת המהדר שתשתמש בה, אינך אמור לקבל כל הודעות שגיאה או אזהרות כתוצאה מהידור דוגמה פשוטה זו.




**תרשים 3-2** הידור HelloWorld.cs אמור להסתיים ללא הודעות שגיאה או אזהרות

אם תקבל הודעת שגיאה כמו זו:

'csc' is not recognized as an internal or external command, operable program or batch file

קרוב לוודאי שלא התקנת את ה-SDK של NET, הכוללת את המהדר של #C.

אם אתה מפעיל את המהדר באופן שגוי – לדוגמה, מספק שם קובץ לא חוקי – או שאתה קורא לו עם ארגומנט ?/, המהדר יפרט את כל אחד מהמתגים (switches) שניתן לקבוע כאשר מהדרים יישומים. איני רוצה לעסוק עכשיו בתיאור כל הדגלים שניתן לקבוע למהדר זה, ולכן אתמקד רק במתגים החשובים לדיון הנוכחי. תוכל לחקור את השאר בזמנך הפנוי.

## הפעלת היישום

כעת, משסיימת לבנות את היישום "Hello, World", הבה נפעיל אותו בכדי לוודא שסביבת זמן הריצה שלך מותקנת בצורה נכונה. כל דוגמאות הספר יהיו "יישומי מסוף", כלומר נישאר צמודים ל-#C ולא נעסוק בדברים ייחודיים לפיתוח ב-Windows. לכן תוכל להפעיל את הדוגמאות האלו משורת הפקודה או מהעורך שלך אם הוא תומך בהפעלת יישומי שורת פקודה.

אם אתה משתמש באפשרות של שורת הפקודה, פתח חלון פקודה והקלד HelloWorld בשורת הפקודה מהספריה שבה בנית את היישום. הפלט אמור להיראות בדומה לזה:

```
d:\>HelloWorld
Hello, World
```

אם אתה מנסה להפעיל את היישום מתוך עורך ואינך רואה דבר, או שחלון פקודה מוקפץ במסך, מפעיל את היישום ונעלם במהירות, שנה את הקוד באופן הבא:

```
class HelloWorld
{
   public static void Main()
   {
      System.Console.WriteLine("Hello, World");
      String str = System.Console.ReadLine();
   }
}
```

הקריאה **System.Console.ReadLine** משהה את היישום עד שתקיש על Enter, וכך תוכל לראות שהיישום אכן מציג את מה שהוא צריך להציג. מכאן ואילך, דוגמאות הספר לא יכללו שורת קוד זו. אם אתה משתמש בעורך להפעלת הדוגמאות, זכור להכניס את השורה לפני סופו של כל יישום.

# סקירת הקוד

לאחר שהנחנו את היסוד לכתיבה וביצוע של יישום #C, הבה נצעד לתוך הקוד ונבחן את המבנה הבסיסי שלו.

## תכנות במקום אחד

כפי שראית בפרקים הקודמים וביישום "Hello, World", השיטות של כל מחלקה הוגדרו בתוך הגדרת המחלקה עצמה. אין זה רק תרגיל של נוחות מבחינתי. כאשר כותבים תוכניות ב- ++C יש למתכנת שתי דרכי פעולה: הכללת הגדרת הפונקציה של חבר מחלקה (class member) בתוך הצהרת המחלקה – דוגמה של **תכנות בקוד כלול** (inline programming) – או הפרדת הצהרת המחלקה מהגדרות פונקציית המחלקה (member function) בקבצים נפרדים. ב-#C אפשרות זו אינה קיימת.

כאשר מגדירים מחלקה ב-#C, חייבים להגדיר בה את כל השיטות; ב- #C אין מנגנון לניהול קבצי כותר (header files). יש תועלת בכך, כי תכנות מסוג זה מאפשר למתכנתי מחלקות ליצור קוד בעל ניידות גבוהה – אחד ממושגי המפתח של סביבת NET.

בדרך זו, כאשר כותבים מחלקה של #C, מקבלים חבילה מכומסת במלואה של פונקציונליות שניתן להציבה בקלות בכל סביבת פיתוח אחרת, בלי לדאוג אם השפה מסוגלת לעבד קבצי include או מנגנון מקביל המאפשר לכלול קבצים בתוך קבצים. השימוש בגישה זו של "תכנות במקום אחד" (one-stop programming) מאפשר לדוגמה, להכניס מחלקה שלמה לתוך דף ASP והיא תתפקד בו, בדיוק באותו אופן כאילו המחלקה צורפה ליישום חלונאי רגיל ועברה הידור.

## Classes ו- Members

הדבר הראשון שאתה רואה ביישום #C בסיסי הוא שם המחלקה, או מרחב של שמות. כפי שלמדת בפרק 1, עליך לבחור שם מחלקה המתאר את מרחב הבעיה, כמו לדוגמה Invoice, PurchaseOrder או Customer. ההתחלה והסוף של הגדרת מחלקה מסומנים בסוגריים מסולסלים: { (פותח) ו- } (סוגר) בהתאמה. כל מה שביניהם נחשב לחלק מהמחלקה. בדוגמה שלנו, שם המחלקה הוא HelloWorld וכל היישום מוגדר בתוך מחלקה זו.

כל חברי המחלקה (class members) מוגדרים בתוך הסוגריים המסולסלים. חברים אלה כוללים שיטות, שדות, מאפיינים, סדרנים (indexers), תכונות וממשקים. בפרקים הבאים אסביר איך מגדירים כל אחד מרכיבי #C השונים האלה.

## השיטה Main

לכל יישום #C חייבת להיות שיטה בשם Main, המוגדרת באחת ממחלקותיו. אין כל חשיבות איזו מחלקה מכילה את השיטה הזו. היישום יכול לכלול מספר רב של מחלקות ובלבד שמחלקה אחת מהן תכיל שיטה בשם Main. חייבים להגדיר access modifier לשיטה זו: public (ציבורית) ו-static (סטטית), protected ואפילו internal. מילת המפתח public היא למעשה מציין גישה (access modifier) המודיע למהדר #C ששיטה זו נגישה מכל קוד. כפי שראית בפרק 1, מילת המפתח static אומרת למהדר שהשיטה Main היא גלובלית, ולכן ניתן לקרוא לה בלי ליצור מופע של המחלקה. לולא הגדרת השיטה, המהדר לא היה יודע איך או מתי ליצור מופע של המחלקה. מכיון שהשיטה היא סטטית, המהדר שומר את כתובת השיטה כנקודת כניסה, כך שמערכת זמן הריצה של NET יודעת היכן להתחיל את ביצוע היישום.

הדוגמאות בפרק זה מציגות את השיטה Main המחזירה void ואינה מקבלת שום ארגומנטים. עם זאת, אתה יכול להגדיר את השיטה Main שלך, כך שתחזיר ערך וגם תקבל מערך של ארגומנטים. אפשרויות אלו, כמו גם הדרך לבצע איטרציה על הארגומנטים המועברים לשיטה Main של היישום, נסקרות בפרק 5.

עוד על מציייני גישה (access modifiers) ראה בפרק 5.

## השיטה System.Console.WriteLine

השיטה System.Console.WriteLine כותבת להתקן הפלט הסטנדרטי את הטקסט המועבר אליה, ומוסיפה לו תו סיום שורה. הפלט מוצג בחלון מסוף (console window), פרט למקרים שבהם אתה משנה זאת, או משתמש בעורך המכוון את הפלט לחלון אחר.

## using Directive-ו Namespaces

בפרק 2 למדת שספריות המחלקה של סביבת NET מאורגנות במבנה היררכי של מרחבי שמות (namespaces). כתוצאה מכך יכולים להתקבל שמות ארוכים למדי כאשר מחלקה או סוג נתונים רצוי, מקוננים מספר שכבות לעומק. לכן, בכדי לחסוך בהקלדה, מספקת #C את ההנחיה using. הבה נמחיש כיצד היא פועלת. בדוגמה "Hello, World" נמצאת שורת הקוד הבאה:

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

כתיבת שורה זו אינה עניין גדול, אך היא עונה על הצורך לאפיין כל סוג נתונים או מחלקה ביישום גדול במלואה. ההנחיה using מאפשרת לספק מעין רשימת נתיבי חיפוש, כך שאם המהדר אינו מבין את המילה שהקלדת, הוא יחפש את ההגדרה ברשימת מרחבי השמות. אם נכלול את ההנחיה using בדוגמה שלנו, הקוד יראה כך:

```
using System;

class HelloWorld
{
   public static void Main()
   {
      Console.WriteLine("Hello, World");
   }
}
```

כאשר המהדר מנתח את השיטה Console.WriteLine, הוא נוכח לדעת שהשיטה אינה מוגדרת. לכן הוא יחפש אז את הגדרת השיטה במרחבי השמות המצוינים עם הנחיות using, וכאשר הוא ימצא אותה במרחב השמות System, הוא יהדר את הקוד ללא שגיאה.

שים לב שההנחיה using חלה על מרחבי שמות, ולא על מחלקות. בדוגמה שלנו, System הוא מרחב השמות, Console היא המחלקה ו-WriteLine היא שיטה השייכת ל-Console.

לפיכך, הקוד הבא אינו חוקי:

```
using System.Console; // ERROR Can't use a using directive with a class.

class HelloWorld
{
   public static void Main()
   {
      WriteLine("Hello, World");
   }
}
```

אף על פי שאינך יכול לציין שם מחלקה בהנחיה using, תוכל ליצור כינויים למחלקות באמצעות תחביר using הבאה:

```
using alias = class
```

אם נשתמש בצורה זו של ההנחיה using, הקוד יהיה:

```
using output = System.Console;

class HelloWorld
{
   public static void Main()
   {
      output.WriteLine("Hello, World");
   }
}
```

דרך זו מספקת את הגמישות להעניק כינויים בעלי משמעות למחלקות שמקוננות מספר שכבות לעומק בהיררכיה של NET, ובכך להפוך את הקוד לקל יותר לכתיבה ולאחזקה.

## Skeleton code

כעת אציג את מה שיכול להיחשב **קוד שלד** (skeleton) כמעט עבור כל יישום #C. קוד זה ממחיש את קווי המתאר עבור יישום #C פשוט. אולי תרצה להקליד את קוד השלד הזה לקובץ ולשמור אותו כתבנית לכתיבת יישום. שים לב שהסוגריים הזוויתיים מציינים היכן עליך לספק מידע.

```
using <namespace>
namespace <your optional namespace>
class <your class>
{
   public static void Main()
   {
   }
}
```

## אי-בהירות של מחלקות

כאשר מגדירים סוג נתונים ביותר ממרחב שמות אחד ויש הפניה למרחבי שמות אלה באמצעות using, המהדר ידפיס הודעת שגיאה המציינת את אי-הבהירות. לפיכך, הקוד הבא לא יעבור הידור מוצלח מכיון שהמחלקה C מוגדרת בשני מרחבי שמות, ואל שניהם יש הפניה באמצעות ההנחיה using:

```
using A;
using B;

namespace A
{
   class C
   {
      public static void foo()
      {
         System.Console.WriteLine("A.C.foo");
      }
   }
}

namespace B
{
   class C
   {
       public static void foo()
      {
         System.Console.WriteLine("B.C.foo");
      }
   }
}

class MultiplyDefinedClassesApp
{
   public static void Main()
   {
      C.foo();
   }
}
```

כדי להימנע משגיאות מסוג זה, הקפד להעניק למחלקות ולשיטות שלך שמות תיאוריים ייחודיים.

# שגיאות ופתרונן

עד כה למדנו איך לכתוב, להדר ולהפעיל יישומי #C, ויש לנו מסגרת בסיסית המשמשת תבנית לבניית יישום #C. עם זאת, מה עושים אם משהו משתבש, כלומר אם קורים דברים לא צפויים? בתכנות נבחין בין שני סוגי תקלות:

- שגיאות זמן-הידור (compile-time errors).
- שגיאות זמן-ריצה (run-time errors).

הבה נתבונן במספר דוגמאות מכל סוג ונלמד איך לתקן אותן.

## Compile-Time Errors

כאשר מהדר , ובכלל זה מהדר #C, אינו יכול לפרש את מה שאתה מנסה להעביר לו – הוא ידפיס הודעת שגיאה ובניית היישום תיכשל. הקלד את הקוד הבא לקובץ בשם HelloErrors.cs והדר אותו:

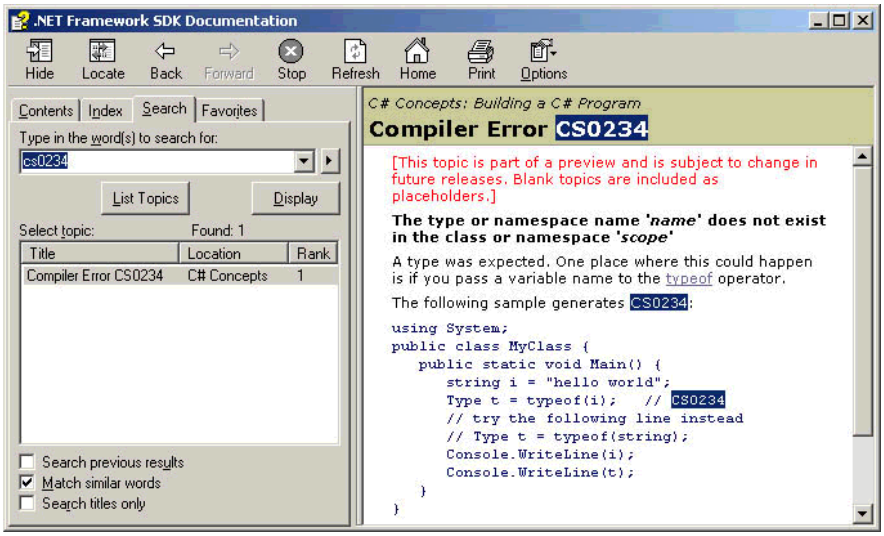
```
using Syste;

class HelloErrors
{
   public static void Main()
   {
      xConsole.WriteLine("Hello, World");
      Console.WriteLinex("Hello, World");
   }
}
```

הפעלת המהדר על קובץ זה תפיק את הפלט הבא:

```
HelloErrors.cs(1,7): error CS0234: The type or namespace name
' Syste' does not exist in the class or namespace ' '
```

הודעה זו מציינת שהמהדר אינו יכול למצוא מרחב שמות בשם Syste, והסיבות מובנות. עם זאת, כוונתי היא להראות למה יש לצפות כאשר המהדר נתקל בשגיאות תחביר בקוד. תחילה מופיע שם הקובץ הנוכחי שעובר הידור ואחר כך מוצגים מספר השורה ומיקום העמודה של השגיאה. לאחר מכן מופיע קוד השגיאה כפי שהמהדר מגדיר אותו – במקרה זה, CS0234.

לבסוף, לאחר קוד השגיאה יש תיאור קצר של השגיאה. בדרך כלל תיאור זה מספיק להבנת השגיאה. אם לא, וכדי לקבל תיאור מפורט יותר, ניתן לחפש את קוד השגיאה בתיעוד של סביבת הפיתוח, המותקן ביחד עם SDK של סביבת NET. העזרה המקוונת המשויכת לקוד השגיאה CS0234 מוצגת בתרשים 3-3.




**תרשים 3-3** תוכל להשתמש בקוד השגיאה שמהדר #C מספק, בכדי למצוא תיאור מפורט שלה בתיעוד העזרה המקוון

שים לב, אף על פי ששתלנו שלוש שגיאות (שגיאות כתיב של מרחב השמות System, המחלקה Console והקריאה לשיטה WriteLine), המהדר מדווח על שגיאה אחת בלבד! זאת, משום ששגיאות מסוימות גורמות למהדר להפסיק את תהליך ההידור מייד כשהוא נתקל בהן, ולהדפיס את השגיאות שהצטברו עד לנקודה זו. בדוגמה זו, המהדר הפסיק את תהליך ההידור כשלא הצליח לפתור את ההנחיה using, משום ששגיאה זו היתה עשויה להיות הסיבה לשגיאות רבות נוספות. לאחר שתאיית נכון את מרחב השמות System בהנחיה using, המהדר ידווח על מספרי השורות ומיקומי העמודות של שתי השגיאות הנותרות.

# ILDASM

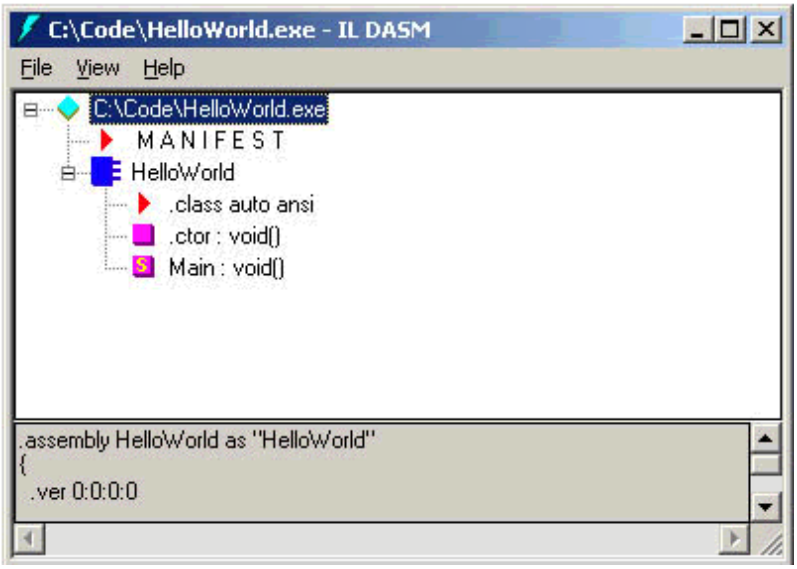
כפי שקראת בפרק 2, כאשר אתה יוצר EXE או DLL באמצעות מהדר NET, הקובץ אינו קובץ הפעלה רגיל. במקום זאת, הוא מכיל מצהר (manifest) המפרט את הסוגים והמחלקות הכלולים בו, וקוד MSIL (Microsoft intermediate language) שהוא קוד פעולה המהודר ומבוצע לאחר מכן באמצעות מהדר מסוג JIT (JITer), על ידי יישום ההתקנה שלך, או על ידי סביבת NET.

קוד MSIL נראה כמו שפת אסמבלי, ולפיכך יכול לשמש ככלי לימוד רב-עוצמה הממחיש מה המהדר עשה בקוד. מסיבה זו, במקומות רבים בספר אשתמש בפלט MSIL של מהדר #C בכדי להמחיש איך פועל משהו מסוים בקרביים של המהדר, או כדי להסביר מדוע עליך להשתמש בתכונה מיוחדת של השפה באופן מסוים.

כדי לראות את פלט MSIL הנוצר על ידי מהדרי NET, מספקת מיקרוסופט NET Framework IL Disassembler. או בקיצור, ILDASM. הוא מאפשר לפתוח קובץ הפעלה של NET (EXE או DLL) ולעיין במרחבי השמות, המחלקות, סוגי הנתונים והקוד שלו. בסעיף הבא נכיר כלי זה מקרוב.

## "Hello, World" ב-MSIL

מתפריט Start פתח את תיבת הדו-שיח Run, הקלד **ildasm** והקש OK. ייפתח חלון יישום חסר ייחודיות, המכיל מספר אפשרויות תפריט. לחץ על תפריט File ובחר Open. עם הופעת תיבת הדו-שיח File Open, נווט לספריה המכילה את היישום HelloWorld.exe שיצרת קודם לכן, ובחר בקובץ זה. כפי שמוצג בתרשים 3-4, הדברים נראים כעת קצת יותר מבטיחים.




**תרשים 3-4** ILDASM מאפשר לחקור את המצהר ואת קוד הפעולה IL המרכיבים כל יישום NET

שים לב למראה העץ שבו משתמש ILDASM לפירוט התכולה של קוד בינארי מנוהל. תרשים 3-5 מציג את הסמלים השונים שבהם משתמש ILDASM לתיאור החלקים של יישום NET. כפי שתוכל לראות מהשוואת סמלים אלה לתוכנית "Hello, World" ב-ILDASM, HelloWorld.exe מכיל מצהר (manifest), מחלקה אחת (HelloWorld), שתי שיטות (בנאי מחלקה והשיטה הסטטית Main), ומעט מידע מחלקה.

| | | |
|---|---|---|
| Namespace: | | (Blue shield) |
| Class: | | (Blue rectangle with three outputs) |
| Interface: | I | (Blue rectangle with three outputs makred 'I') |
| Value class: | | (Brown rectangle with three outputs) |
| Enum: | E | (Brown rectangle with three outputs marked 'E') |
| Method: | | (Magenta rectangle) |
| Static method: | S | (Magenta rectangle marked 'S') |
| Field: | | (Cyan diamond) |
| Static field: | S | (Cyan diamond marked 'S') |
| Event: | | (Green point-down triangle) |
| Property: | | (Red point-up triangle) |
| Manifest or a class info item: | | (Red point-right triangle) |

**תרשים 3-5** הסמלים השונים המשמשים לציון החלקים של יישום NET ב-ILDASM

החלק המעניין ביותר של "Hello, World" הוא השיטה Main. לחץ לחיצה כפולה על השיטה Main בעץ של ILDASM, והוא יפתח חלון המציג את MSIL של השיטה Main, כמוצג בתרשים 3-6.

```
HelloWorld::Main : void()
.method public hidebysig static void Main() il managed
{
  .entrypoint
  // Code size       11 (0xb)
  .maxstack  8
  IL_0000:  ldstr      "Hello, World"
  IL_0005:  call       void [mscorlib]System.Console::WriteLine(class System
  IL_000a:  ret
} // end of method HelloWorld::Main
```

**תרשים 3-6** כדי לראות קוד MSIL שנוצר עבור שיטה כלשהי, פתח את הקוד הבינארי ב-ILDASM ולחץ לחיצה כפולה על שם השיטה

היישום "Hello, World" אינו נראה מלהיב אפילו ב-MSIL, אבל תוכל ללמוד מספר עובדות אודות קוד MSIL המיוצר עבור כל יישום NET. לשם כך, הבה נבחן את השיטה שורה אחר שורה.

```
.method public hidebysig static void Main() il managed
{
   .entrypoint
   // Code size    11 (0xb)
   .maxstack 8
   IL_0000:   ldstr     "Hello, World"
   IL_0005:   call void [mscorlib]System.Console::WriteLine(class System.String)
   IL_000a:   ret
} // end of method HelloWorld::Main
```

השורה הראשונה מגדירה את השיטה Main על ידי שימוש במילת המפתח "method." של MSIL. נוכל גם לראות שהשיטה מוגדרת כ-public ו-static, שהם מצייני הגישה של ברירת המחדל עבור השיטה Main. אולם בנוסף לכך, אנו רואים ששיטה זו מוגדרת כ-managed (מנוהלת). זו הבחנה חשובה, כי ב-#C אתה גם יכול לכתוב קוד שהוא unmanaged (לא-מנוהל), או unsafe (לא-בטוח). פרק 17 עוסק בקוד #C לא-מנוהל.

שורת הקוד הבאה משתמשת במילת המפתח "entrypoint." של MSIL בכדי לציין ששיטה מסוימת זו היא נקודת הכניסה של היישום. כאשר זמן הריצה של NET מפעיל את היישום, בקרת הריצה עוברת לנקודה זו של התוכנית.

הפריטים המעניינים הבאים הם קוד הפעולה MSIL הממשי בשורות IL_0000 ו-IL_0005. השורה הראשונה משתמשת בקוד הפעולה ldstr (Load String) כדי לטעון מלל המקודד באופן קשיח ("Hello, World") לתוך מחסנית. השורה הבאה קוראת לשיטה System.Console.WriteLine. שים לב ש-MSIL מוסיף לשם השיטה קידומת, המציינת את שם מכלל הקוד (assembly) המגדיר את השיטה. הודות לרמת פירוט כזו של MSIL אפשר לכתוב ביתר קלות כלים עוקבי תלות (dependency walkers), שיכולים להסתכל לתוך יישום כדי לקבוע איזה קבצים נחוצים לו כדי שיפעל בצורה תקינה. בנוסף, אתה יכול לראות את המספר וסוג הארגומנטים שהשיטה מצפה לקבל. במקרה זה, השיטה System.Console.WriteLine תצפה לכך שאובייקט System.String יהיה במחסנית כאשר היא תיקרא. לבסוף, השורה IL_000a היא למעשה הוראת ret (מהמילה return) של MSIL לחזור מהשיטה.

ILDASM הוא כלי רב-עוצמה, ולכן מומלץ מאד להיעזר בו בכל פעם שאני מתייחס לקוד MSIL המיוצר על ידי מהדר #C.

הערה

כדי לדעת אם EXE או DLL הוא מנוהל, נסה לפתוח אותו עם ILDASM. אם זהו קובץ מנוהל חוקי המכיל MSIL ומצהר, הוא ייפתח. אם הקובץ אינו כזה, תקבל את הודעת השגיאה הזו:

<your file> has no valid CLR header and cannot be disassemble

# קווים מנחים לתכנות ב-#C

אסיים את הפרק בתיאור מספר קווים מנחים לכתיבת יישומי #C.

## מתי להגדיר מרחבי שמות משלך

ביישום "Hello, World" השתמשנו בשיטה Console.WriteLine המוגדרת במרחב השמות System. למעשה, כל סוגי הנתונים והמחלקות של NET מוגדרים במרחבי שמות (namespaces). עם זאת, עד כה לא יצרנו מרחב שמות משלנו ולכן נעסוק כעת בנושא זה.

מרחבי שמות הם דרך מצוינת להקבצת סוגי הנתונים והמחלקות שלך על פי קטגוריות ובכך למנוע התנגשויות של שמות. מיקרוסופט הציבה את כל הגדרות המחלקות וסוגי הנתונים של NET במרחבי שמות מוגדרים, מתוך רצון להבטיח שהשמות שלה לא יתנגשו עם השמות של מי שמשתמש במהדרים שלה. עם זאת, הצורך להשתמש במרחבי שמות תלוי בתשובה לשאלה אחת: האם סוגי הנתונים והמחלקות שאתה יוצר ישמשו בסביבה שאינה נשלטת על ידך? במילים אחרות, אם הקוד שלך משמש רק חברים בצוות שלך, תוכל לקבוע בקלות כללים לכתיבת שמות, כך שלא יהיו התנגשויות. עם זאת, כאשר אתה כותב מחלקות שישמשו מפתחים של צד שלישי, מקרה שבו אין לך שליטה על נוהגי קביעת השמות, עליך להשתמש בוודאות במרחבי שמות. מיקרוסופט ממליצה להשתמש בשם החברה שלך כמרחב שמות ברמה העליונה. אני אף ממליץ להשתמש במרחבי שמות בכל זמן שמישהו אחר עשוי לראות את הקוד שלך.

## קווים מנחים לבחירת שמות

הסטטיסטיקה מראה שההוצאה הגדולה ביותר הקשורה לפיתוח יישומים היא התחזוקה. לפני שנמשיך, הייתי רוצה להתעכב מעט על מוסכמות בקביעת שמות, כי קביעת מוסכמות יציבות וקלות להבנה יאפשרו לך לכתוב קוד שהוא קל יותר לקריאה, ולכן – גם לתחזוקה.

כפי שרובנו יודעים, מוסכמות בקביעת שמות הן נושא רגיש. כאשר Visual C++ ו-MFC היו בתחילת דרכן, היה קל יותר להתמודד עם הנושא הזה. דעתי בעניין היתה, שיש להשתמש במוסכמות שבהם השתמשה מיקרוסופט בכתיבת MFC.

אחרי הכל, השימוש בשתי מערכות של קביעת שמות בקוד המקור אינו מועיל כלל. עם זאת, בסביבה החדשה של #C אין רואים את הקוד של מיקרוסופט. למרות זאת, לאחר שיחות רבות עם צוות הפיתוח של #C, הבנתי שמתפתח תקן חדש. קרוב לוודאי שהתקן עשוי להיות מעט שונה ממה שאני מציג כאן, אך ההצעה שלפניך יכולה לשמש לך כנקודת התחלה.

## תקני מוסכמות לקביעת שמות

לפני שאסביר היכן ואיך לקרוא לאלמנטים השונים של היישום, הבה נעיף מבט בתקנים השונים הידועים כיום.

### Hungarian Notation

הסימון ההונגרי (Hungarian notation) הוא המערכת המשמשת את רוב המפתחים ב-C ו- ++C (כולל אלה במיקרוסופט). זו מערכת שלמה לקביעת שמות שפותחה על ידי מהנדס מיקרוסופט בשם Charles Simonyi. בתחילת שנות ה-80, אימצה מיקרוסופט את מערכת הכללים הידועה – או הידועה לשמצה, כפי שטוענים אחדים – על בסיס הרעיונות שהעלה Simonyi בתזת הדוקטורט שלו "Meta-Programming: A Software Production Method".

הסימון ההונגרי קובע שיש להוסיף תחילית (prefix) לכל משתנה לציון הסוג שלו. עם זאת, לא לכל סוג נקבעה תחילית סטנדרטית. בנוסף לכך, כשיצאו לשוק שפות חדשות והוגדרו סוגי נתונים חדשים, היה צורך ליצור תחיליות חדשות, מספרן גדל וחלק מהן לא היו מוכרות למפתחים (אגב, שיטת הסימון נקראת Hungarian notation בגלל העובדה שהתחיליות גורמות למשתנים להיראות כאילו נכתבו בשפה השונה מאנגלית; וגם מוצאו של Simonyi היה מהונגריה).

מיקרוסופט עודדה את השימוש בסימון ההונגרי, על ידי הספר Programming Windows מאת Charles Petzold (שיצא לאור על ידי Microsoft Press), שהשתמש בניב של הסימון ההונגרי בכל הדוגמאות. בנוסף לכך, מיקרוסופט השתמשה בסימון זה בפיתוחים של עצמה. כאשר מיקרוסופט שיווקה את MFC היא הוסיפה לקוד המקור צרור של תחיליות חדשות ייחודיות לפיתוח של ++C, ובכך הבטיחה את המשכיות השימוש בסימון ההונגרי.

ובכן, מדוע לא להמשיך להשתמש בסימון ההונגרי? – משום שהסימון ההונגרי שימושי במצבים שבהם יש תועלת בידיעת הסוג, או טווח ההכרה של המשתנה. עם זאת, כפי שתראה ביתר פירוט בפרק הבא, כל סוגי הנתונים ב-#C הם אובייקטים ומבוססים על המחלקה System.Object של NET. לכן, לכל המשתנים יש קבוצה בסיסית של פונקציונליות ומאפיינים התנהגותיים. מסיבה זו, אין צורך בסימון הונגרי בסביבת NET.

הערה

עבור כל הסקרנים, או עבור אלה הסובלים מנדודי שינה, המאמר המקורי שהציג את הסימון ההונגרי נמצא באתר האינטרנט:

.http://msdn.microsoft.com/library/techart/hunganotat.htm

## Camel Casing-ו Pascal Casing

למרות שלא ניתן היה להכתיב לצוות #C תקן "קשיח", ברור מדרך הכתיבה שהצוות אימץ את סימונו של עובד מיקרוסופט בשם Rob Caron, שהציע להשתמש בתערובת של סימון פסקל (Pascal casing) וסימון גמל (camel casing) בקביעת שמות למשתנים. במאמרו "Coding Techniques and Programming Practices" שהופיע ב-MSDN (http://msdn.microsoft.com/library/techart/cfr.htm), מציע Rob להשתמש בסימון פסקל (כל השמות מתחילים באות רישית) עבור שמות של שיטות ובסימון גמל (כמו סימון פסקל, אך רק החל מהמילה השנייה) עבור שמות משתנים. בהתאם לכך, החלטתי להשתמש באותן המוסכמות בדוגמאות הקוד בספר זה. עם זאת, מכיון ש-#C מכילה אלמנטים נוספים מלבד שיטות ומשתנים, אפרט בסעיפים הבאים את האלמנטים השונים ואת המוסכמות לקביעת שמות בהתאם לשימוש הפנימי של מיקרוסופט אשר בחרתי.

הערה

מידע נוסף בנושא זה תוכל למצוא בקווים המנחים של סביבת NET הנכללים בתיעוד SDK של סביבת NET:

.NET Framework Developer Specifications\.NET Framework Design Guidelines\Naming Guidelines

## Namespaces

השתמש בשם החברה או המוצר שלך, והשתמש בסימון מעורב עם אות תחילית רישית – לדוגמה, *Microsoft*. אם אתה עוסק במכירת רכיבי תוכנה, צור מרחב שמות ברמה העליונה על פי שם החברה שלך, ואז, עבור כל מוצר צור מרחב שמות מקונן עם סוגי הנתונים המשובצים שלו, בכך תמנע התנגשות שמות עם מוצרים אחרים. דוגמה לכך ניתן למצוא ב-SDK של סביבת NET: Microsoft.Win32. אסטרטגיה זו עלולה להביא לקביעת שמות מייגעים להקלדה, אך זכור שמשתמשי הקוד שלך צריכים רק לציין את ההנחיה using כדי לחסוך בהקלדה. אם החברה שלך נקראת Trey Research, ואתה מוכר שני מוצרים, תוכל לכנות את מרחבי השמות שלך: TreyResearch.Grid ו- TreyResearch.Database.

## Classes

אובייקטים אמורים לייצג יישויות אמיתיות בעלות יכולות, ולכן תוכל לקרוא למחלקות בשמות עצם המתארים את מרחב הבעיה של המחלקה. במקרים שבהם המחלקה היא מסוג כללי יותר (כלומר, פחות מסוימת למרחב הבעיה) – לדוגמה, סוג המייצג מחרוזת SQL – השתמש בסימון פסקל (כל השמות מתחילים באות רישית).

## Methods

השתמש בסימון פסקל עבור כל השיטות. שיטות מציינות פעולה ומפיקות תוצאה. לפיכך, השמות שאתה קובע לשיטות צריכים לתאר **מה הן** עושות. דוגמאות לכך הן PrintInvoice ו-OpenDatabase.

במקרה של שיטות שישמשו בביטויים בוליאניים, הקדם לשם השיטה פועל, המציין מה השיטה תעשה. לדוגמה, אם השיטה מחזירה ערך בוליאני כאשר תחנת העבודה נעולה, קבע לשיטה שם, כמו למשל IsWorkStationLocked. בדרך זו, אם השיטה רשומה במשפט If, משמעותה נעשית ברורה יותר, כמו כאן:

```
if (IsWorkStationLocked) ...
```

## Method Arguments

השתמש בסימון פסקל עבור כל הארגומנטים. קבע לארגומנטים שמות בעלי משמעות, כך שכאשר אתה כותב IntelliSense, המשתמש יכול להבין בקלות למה משמש כל ארגומנט.

## Interfaces

השתמש בסימון פסקל עבור כל הממשקים. מקובל לצרף לשם הממשק את האות "I" – לדוגמה, Icomparable (זו המוסכמה היחידה שאני מכיר ב-#C, אשר דומה לסימון ההונגרי).

מפתחים רבים משתמשים באותם הכללים לקביעת שמות עבור ממשקים ועבור מחלקות. עם זאת, קיים הבדל פילוסופי יסודי בין השניים. מחלקות מייצגות כימוס של הנתונים והפונקציות הפועלות עליהם. ממשקים מייצגים התנהגות, ויישום הממשק מאפשר למחלקה לחשוף התנהגות זו. לכן, מקובל לקבוע לממשקים שמות תואר. לדוגמה, ממשק שמצהיר על שיטות לרישום נתונים עשוי להיקרא ISerializable כדי לציין את היכולת של המחלקה המיישמת.

### Class Members

זה אולי הנושא הבעייתי ביותר בנוגע למפתחי #C. אלה מאתנו שהגיעו מ- ++C ו-MFC מורגלים להוסיף לשמות החברים את הקידומת _m. בכל זאת, אני ממליץ להשתמש בסימון גמל (camel casing), שבו האות הראשונה אינה רישית. בדרך זו, אם יש שיטה המקבלת ארגומנט כמו Foo, תוכל להבדילו מהייצוג הפנימי של משתנה זה על ידי יצירת חבר פנימי בשם foo.

הקדמת שם המחלקה לשם של חבר מיותרת. נניח ששם המחלקה הוא Author. במקום ליצור חבר בשם AuthorName, אשר ייכתב לאחר מכן כ- Author.AuthorName, קבע לחבר את השם Name.

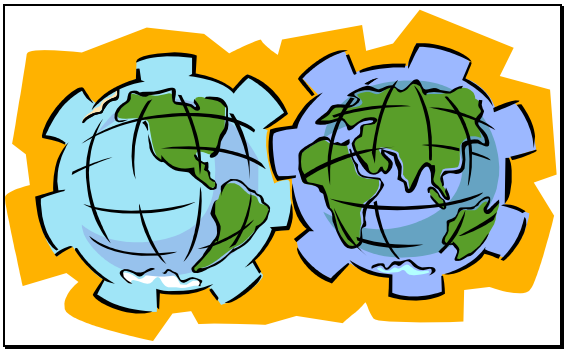
# סיכום

צעד ראשון חשוב בלימוד #C הוא לדעת איך לכתוב, להדר ולהפעיל תוכניות #C. בדרך כלל אין חשיבות לעורך המשמש לעריכת קבצי מקור של #C, אך יש יתרונות לשימוש בסביבות ובעורכים חזקים המותאמים לפיתוח בשפת #C. הכרת מיגוון האפשרויות והמתגים של מהדר #C מאפשרת לשלוט באופן שבו יוצר המהדר קוד MSIL. תוכל לחקור קוד זה בעזרת כלים כמו ILDASM, הכלול ב-SDK של סביבת NET.

תוכניות #C הן בעלות מבנה המספק מספר תכונות שהופכות את התוכניות לקלות ובטוחות יותר לכתיבה, ומועדות פחות לשגיאות. אוסף תכונות זה כולל את מרחבי השמות ואת ההנחיה using. לסיום, מוסכמות לקביעת שמות, הכוללות מוסכמות מוגדרות לשימוש באותיות רישיות, יכולות להפוך תוכניות לקריאות יותר, מובנות יותר וגם קלות יותר לתחזוקה.

## חלק 2

# יסודות המחלקה של #C



# פרק 4

# CTS - Common Type System

בליבה של סביבת העבודה NET נמצאת מערכת סוגי נתונים אוניברסלית הנקראת **CTS** (Common Type System). בנוסף להגדרת כל הסוגים, CTS קובעת את הכללים שסביבת זמן הריצה CLR פועלת לפיהם, בכל הקשור להכרזה ולשימוש של יישומים בסוגים אלה. בפרק זה נסקור את מערכת הסוגים החדשה, כדי שתכיר את הסוגים הזמינים למפתחי #C ותבין את ההשלכות של השימוש בסוגים השונים בתוכניות #C. תחילה נחקור את משמעות התפישה האומרת שכל אלמנט תכנותי הוא אובייקט ב-NET. בהמשך נלמד איך NET מחלקת את הסוגים לשתי קטגוריות:

- value types (סוגי ערכים).
- reference types (סוגי ייחוס).

נגלה איך אריזה מאפשרת למערכת סוגים מוכוונת-עצמים שלמה לתפקד ביעילות. בנוסף, נבחן איך פועלת המרת סוגים ב-#C ואת השימוש במרחבי שמות.

## כל דבר הוא אובייקט

רוב השפות מוכוונות-העצמים מכילות שתי קבוצות נפרדות של סוגי נתונים: primitive types (סוגים בסיסיים) שהם חלק מהשפה, וסוגים הניתנים ליצירה על ידי משתמשי השפה – מחלקות (classes). primitive types הם בדרך כלל סוגי נתונים פשוטים כמו תווים, מחרוזות ומספרים, בעוד שמחלקות הן סוגים הנוטים להיות מורכבים יותר.

קיומן של שתי קבוצות סוגי נתונים נפרדות מהווה מקור לבעיות רבות. בעיה אחת קשורה לתאימות. לדוגמה, כדי לקבל אוסף של ערכים מסוג int במערכת מסורתית עם שתי קבוצות של סוגים, צריך ליצור מחלקה באופן מיוחד. כאשר רוצים אוסף של ערכים מסוג double צריך לעשות אותו הדבר עבור סוג זה. הסיבה לכך היא, שבאופן רגיל אין ל-primitive types אלה שום דבר משותף.

הם אינם אובייקטים אמיתיים, ולכן אינם נגזרים ממחלקת בסיס משותפת. כל סוג צריך להיות מטופל באופן ייחודי על פי מונחיו. בעיה דומה מתעוררת כאשר רוצים לציין ששיטה יכולה לקבל ארגומנטים **מכל** סוג הנתמך על ידי השפה. מכיון ש-primitive types אלה אינם תואמים זה לזה, לא ניתן לציין ארגומנט כזה, אלא אם כותבים מחלקות עטיפה (wrapper classes) לכל primitive type.

# Reference Types-ו Value Types

כתיבת שפה שבה כל אלמנט הוא אובייקט אינה רעיון חדש. שפות אחרות ניסו זאת, כאשר המפורסמת שבהן היא SmallTalk. החיסרון הגדול ביותר בגישה זאת היה תמיד ביצועים ירודים. לדוגמה, כאשר אתה מנסה לחבר שני ערכים מסוג double ב- SmallTalk, מתבצעת הקצאת אובייקט. מיותר לציין שהקצאה זו מאד לא יעילה כאשר רוצים רק לסכם שני מספרים. הבזבוז נובע מזמן התקורה לכל הפעולות המיותרות במקרה זה ובדומיו.

מתכנני CTS עמדו בפני המשימה ליצור מערכת סוגים שבה כל דבר הוא אובייקט וגם היא עדיין פועלת ביעילות. הפתרון שלהם היה להפריד את סוגי CTS לשתי קטגוריות:

- **value types (סוגי ערכים).**
- **reference types (סוגי ייחוס).**

מונחים אלה, כפי שתראה מייד, משפיעים על אופן ההקצאה של משתנים ועל אופן תפקודם הפנימי.

## Value Types

משתנה מסוג value מכיל נתונים ממשיים. מכאן, הכלל הראשון של value types הוא שהם אינם יכולים להיות null. כאן לדוגמה, אני מקצה זיכרון על ידי יצירת משתנה מסוג System.Int32 של CTS. הצהרה זו יוצרת הקצאת מחסנית, בגודל של 32 סיביות.

```
int i = 32;
```

בנוסף לכך, כתוצאה מההשמת ערך ל-i, מועבר ערך של 32 סיביות למקום שהוקצה במחסנית.

בשפת #C מוגדרים מספר value types, כולל מונים (enumerators), מבנים (structures) ו-primitive types . כאשר אתה מצהיר על אחד מהסוגים האלה, אתה מקצה במחסנית את מספר הבתים המשויך לאותו סוג ופועל ישירות עם מערך מוקצה זה של סיביות. בנוסף, כאשר אתה מעביר משתנה מסוג value, אתה מעביר אל האובייקט את ערך המשתנה, ולא את הייחוס שלו.

## Reference Types

reference types דומים למה שמקובל ב-++C בכך שהם מצביעים מבטיחי-סוג (type-safe). תכונה זו פירושה שייחוס אינו רק כתובת שאולי מצביעה על מה שאתה חושב, אלא הוא תמיד מבטיח (כל זמן שאינו null) שהוא מצביע לאובייקט מהסוג המוגדר ואשר כבר הוקצה לו מקום במצבור הזיכרון (heap). שים לב גם לעובדה ש-reference יכול לקבל את הערך null.

בדוגמה הבאה הוקצה reference type מסוג string, אך מבחינה פנימית הוקצה ערך במצבור הזיכרון והוחזר ייחוס (הפניה) לערך זה.

```
string s = "Hello, World";
```

כמו ב-value types, יש ב- #C מספר סוגים המוגדרים כ-reference types: מחלקות (classes), מערכים (arrays), נציגים (delegates) וממשקים (interfaces). בכל פעם שאתה מצהיר על משתנה מאחד מן הסוגים האלה, אתה מקצה במצבור הזיכרון את מספר הבתים המשויכים לאותו סוג, ופועל עם הפניה לאותו אובייקט במקום לפעול ישירות בסיביות (כמו ב-value types).

## Unboxing-ו Boxing

כיצד קטגוריות שונות של סוגים עושות את המערכת ליותר יעילה? הדבר נעשה באמצעות הקסם הנקרא **אריזה** (boxing). במובן הפשוט, אריזה היא המרה של value type ל-reference type. המקרה ההפוך הוא **פריקה** (unboxing) של reference type ל-value type.

הדבר הטוב בטכניקה זו הוא שאובייקט נחשב אובייקט רק כאשר הוא צריך להיות כזה. לדוגמה, אם אתה מצהיר על משתנה מסוג value כמו System.Int32, מוקצה עבורו זיכרון במחסנית. אתה יכול להעביר את המשתנה הזה לכל שיטה המוגדרת כמקבלת משתנה מסוג System.Object, ואתה יכול לגשת לכל אחד מהחברים שלו, עבורם יש לך גישה. לפיכך משתנה זה נראה כאובייקט לכל דבר, כאשר למעשה מדובר בארבעה בתים במחסנית בסך הכל.

המערכת אורזת את המשתנה באופן אוטומטי רק כאשר אתה מנסה להשתמש בו באופן התואם לממשק מחלקת הבסיס System.Object המשתמע שלו, וכך הוא הופך ל-reference type שניתן להשתמש בו כמו בכל אובייקט. אריזה היא הדרך שבה #C מאפשרת לכל דבר להיראות כמו אובייקט, ובכך להימנע מהתקורה שהיתה נדרשת אילו כל דבר היה אובייקט באמת. נבחן מספר דוגמאות להבנת מושגים אלה.

```
int foo = 42;          //  Value type.
object bar = foo;      //  foo is boxed to bar.
```

בשורת הקוד הראשונה אנו יוצרים משתנה (foo) מסוג int. הסוג int הוא כידוע value type (מכיון שהוא primitive type). בשורה השנייה, המהדר מבחין שהמשתנה foo מועתק ל-reference type, המיוצג על ידי המשתנה bar, ויוצר את קוד MSIL הדרוש לאריזת ערך זה.

כדי להפוך חזרה את המשתנה bar ל-value type, תוכל לבצע המרה מפורשת:

```
int foo = 42;          //  Value type.
object bar = foo;      //  foo is boxed to bar.
int foo2 = (int)bar;   //  Unboxed back to int.
```

שים לב שאריזה, כלומר המרה מ-value type ל-reference type, אינה צריכה להיות מפורשת. אולם בפריקה, כלומר בהמרה מ-reference type ל-value type, חייבים לציין איזה value type רוצים לחלץ. הסיבה לכך היא שבמקרה של פריקה ניתן להמיר אובייקט לכל סוג. לכן, המרה מפורשת נחוצה כדי שהמהדר יוכל לוודא שההמרה חוקית עבור סוג המשתנה המוגדר. מכיון שהמרה כרוכה בחוקים נוקשים ומכיון שחוקים אלה נשלטים על ידי ה- CTS, נסקור נושא זה בהרחבה בהמשך הפרק, כשנעסוק בהמרות.

# המקור לכל הסוגים: System.Object

כל הסוגים נגזרים מהסוג System.Object, ובכך מובטח שלכל סוג במערכת יש קבוצת יכולות מינימלית. טבלה 4-1 מתארת את ארבע השיטות הציבוריות שכל הסוגים מקבלים בחינם.

**טבלה 4-1** שיטות ציבוריות של הסוג System.Object

| שם השיטה | תיאור |
|---|---|
| bool Equals() | שיטה זו משווה ייחוסים של שני אובייקטים בזמן ריצה כדי לקבוע אם יש זהות ביניהם. אם שני המשתנים מתייחסים לאותו אובייקט, הערך המוחזר הוא true. במקרה של value types, הנידונים בסעיף הבא, שיטה זו מחזירה true כאשר שני הסוגים זהים ובעלי אותו ערך. |
| int GetHashCode() | מחזירה את hash code המוגדר עבור אובייקט. פונקציות hash מנוצלות כאשר מי שמיישם מחלקה רוצה להציב קוד hash של אובייקט בטבלת hash משיקולי ביצוע. |
| Type GetType() | משמשת ביחד עם שיטות שיקוף (reflection, הנידונות בפרק 16) לאחזור מידע על הסוג עבור אובייקט נתון. |
| string ToString | כברירת מחדל, שיטה זו משמשת לקבלת שם האובייקט. ניתן לדרוס אותה על ידי מחלקות נגזרות כך שתחזיר שם מייצג ידידותי למשתמש של אובייקט. |

טבלה 4-2 מפרטת את השיטות המוגנות של System.Object.

**טבלה 4-2** שיטות מוגנות של הסוג System.Object

| שם השיטה | תיאור |
|---|---|
| void Finalizer() | שיטה זו נקראת על ידי סביבת זמן הריצה, כדי לאפשר ניקוי לפני garbage collector (אוסף האשפה). שים לב שהקריאה לשיטה זו לא מתבצעת תמיד. לפיכך, אל תכניס קוד שחייב להתבצע בתוך שיטה זו. כלל זה נוגע לעניין הנקרא סיום דטרמיניסטי (deterministic finalization) שנדון בו בפירוט בפרק 5. |
| Object MemberwiseClone | חבר זה מייצג עותק שטחי (shallow copy) של האובייקט, כלומר, עותק של האובייקט המכיל ייחוסים לאובייקטים אחרים ואשר אינו מכיל עותק שלהם. אם המחלקה שלך צריכה לתמוך בעותק עמוק (deep copy), הכולל עותקים של האובייקטים המיוחסים, עליך ליישם את הממשק ICloneable ולבצע את השכפול, או ההעתקה, באופן ידני בעצמך. |

# Aliases-ו Types

בעוד שה-CTS אחראית להגדרת הסוגים הניתנים לשימוש בשפות NET, רוב השפות בוחרות לתת כינויים (aliases) לסוגים אלה. לדוגמה, ערך Integer בגודל 4 בתים מיוצג על ידי הסוג System.Int32 של CTS. שפת C# מגדירה לסוג זה את הכינוי int. אין כל יתרון לשימוש בכל אחת מהטכניקות על האחרת. טבלה 4-3 מפרטת את סוגי CTS השונים ואת כינוייהם ב- C#.

**טבלה 4.3** סוגי וכינויי CTS

| שם סוג CTS | כינוי C# | תיאור |
|---|---|---|
| System.Object | object | מחלקת בסיס עבור כל סוגי CTS |
| System.String | string | String (סוג נתונים המאפשר טיפול במחרוזות) |
| System.Sbyte | sbyte | Signed 8-bit byte (מייצג ערך שלם עם סימן באמצעות 8 סיביות) |
| System.Byte | byte | Unsigned 8-bit byte (מייצג ערך שלם ללא סימן באמצעות 8 סיביות) |
| System.Int16 | short | Signed 16-bit value (מייצג ערך שלם עם סימן באמצעות 16 סיביות) |

| שם סוג CTS | כינוי #C | תיאור |
|---|---|---|
| System.Uint16 | ushort | Unsigned 16-bit byte (מייצג ערך שלם ללא סימן באמצעות 16 סיביות) |
| System.Int32 | int | Signed 32-bit value (מייצג ערך שלם עם סימן באמצעות 32 סיביות) |
| System.Uint32 | uint | Unsigned 32-bit value (מייצג ערך ללא סימן באמצעות 32 סיביות) |
| System.Int64 | long | Singed 64-bit value (מייצג ערך שלם עם סימן באמצעות 64 סיביות) |
| System.Uint64 | ulong | Unsigned 64-bit value (מייצג ערך שלם ללא סימן באמצעות 64 סיביות) |
| System.Char | char | 16-bit Unicode (תו Unicode באורך 16 סיביות) |
| System.Single | float | IEEE 32-bit float (מייצג ערך מסוג נקודה צפה באמצעות 32 סיביות בדיוק של 7 ספרות) |
| System.Ddouble | double | IEEE 64-bit float (מייצג ערך מסוג נקודה צפה באמצעות 64 סיביות בדיוק של 15-16 ספרות) |
| System.Boolean | bool | ערך בוליאני (true/false) |
| System.Decimal | decimal | סוג נתונים בגודל של 128 סיביות עם דיוק עד 28 או 29 מקומות עשרוניים – משמש בעיקר ליישומים פיננסיים, שבהם דרושה מידה גבוהה של דיוק. |

# Casting Between Types

**המרה** (casting) היא אחד ההיבטים החשובים ביותר של סוגי נתונים. נניח שקיימת מחלקת בסיס בשם Employee ומחלקה נגזרת בשם ContractEmployee. הקוד הבא תקין, כי תמיד יש המרה משתמעת כלפי מעלה (implied upcast) ממחלקה נגזרת אל מחלקת הבסיס שלה:

```
class Employee { }

class ContractEmployee : Employee { }

class CastExample1
{
   public static void Main ()
   {
      Employee e = new ContractEmployee();
   }
}
```

אולם, הקוד הבא אינו חוקי, כי המהדר אינו יכול לספק המרה משתמעת כלפי מטה (implied downcast):

```
class Employee { }

class ContractEmployee : Employee { }

class CastExample2
{

   public static void Main ()
   {
      ContractEmployee ce = new Employee(); // Won't compile.
   }
}
```

הסיבה להתנהגות השונה מחזירה אותנו לפרק 1, העוסק ביסודות של תכנות מוכוון-עצמים, ולמושג של יכולת המרה (substitutability). להזכירך, הכללים של יכולת המרה קובעים שמחלקה נגזרת יכולה לשמש תחליף למחלקת הבסיס שלה. לפיכך, אובייקט מסוג ContractEmployee צריך תמיד להיות מסוגל להחליף אובייקט Employee. זו הסיבה לכך שהדוגמה הראשונה עוברת הידור מוצלח.

בניגוד לכך, אינך יכול להמיר אובייקט מסוג Employee כלפי מטה לאובייקט מסוג ContractEmployee, כי אין ערובה לכך שהאובייקט תומך בממשק המוגדר על ידי המחלקה ContractEmployee. לכן, במקרה של המרה כלפי מטה, חייבים להשתמש בהמרה מפורשת (explicit upcast), הנה כך:

```
class Employee { }

class ContractEmployee : Employee { }

class CastExample3
{
   public static void Main ()
   {
      //   Downcast will fail.
      ContractEmployee ce = (ContractEmployee)new Employee();
   }
}
```

מה יקרה אם ננסה להערים על ה-CTS על ידי המרה מפורשת של מחלקת בסיס למחלקה נגזרת (derived class), ונעשה זאת כך:

```
class Employee { }

class ContractEmployee : Employee { }

class CastExample4
{
```

```
   public static void Main ()
   {
      Employee e = new Employee();
      ContractEmployee c = (ContractEmployee)e;
   }
}
```

התוכנית עוברת הידור מוצלח, אך הפעלתה יוצרת חריג בזמן ריצה (run-time exception). כאן יש לשים לב לשני דברים: ראשית, לא נוצרת שגיאה בזמן הידור התוכנית, כי e עשוי באמת להיות המרה כלפי מעלה של אובייקט ContractEmployee. לכן, טבעו האמיתי של האובייקט עם המרה כלפי מעלה אינו ידוע עד לביצוע התוכנית. שנית, CLR קובעת את סוגי האובייקטים בזמן ריצה. כאשר CLR מזהה המרה שאינה חוקית, היא מעוררת חריג System.InvalidCastException.

קיימת דרך אחרת להמרת אובייקטים: השימוש במילת המפתח as. לשימוש במילת מפתח זו יש יתרון בכך שאם המרה אינה חוקית, היא אינה מעוררת חריג. הפעולה תפיק ערך null. הנה דוגמה:

```
using System;

class Employee { }

class ContractEmployee : Employee { }

class CastExample5
{
   public static void Main ()
   {
      Employee e = new Employee();
      Console.WriteLine("e = {0}",
                  e == null ? "null" : e.ToString());

      ContractEmployee c  = e as ContractEmployee;
      Console.WriteLine("c = {0}",
                     c == null ? "null" : c.ToString());
   }
}
```

אם אתה מפעיל תוכנית זו, תקבל את התוצאות הבאות:

```
c:>CastExample5
e = Employee
c = null
```

היכולת להשוות אובייקט ל-null פירושה שאינך צריך להסתכן בהתייחסות לאובייקט null. למעשה, אם הדוגמה היתה מנסה לקרוא לשיטה System.Object ולהפעילה על האובייקט c, ה-CTS היה מעורר את החריג System.NullReferenceException.

# Namespaces

מרחבי שמות (namespaces) משמשים להגדרת טווח הכרה ביישומי #C. הצהרה על מרחב שמות מאפשרת למפתח לספק ליישום #C מבנה היררכי המבוסס על קבוצות של סוגים ומרחבי שמות (מקוננים) אחרים בעלי קשר סמנטי. קבצי קוד מקור מרובים יכולים לתרום לאותו מרחב שמות. כך, אם אתה אורז מספר מחלקות במרחב שמות נתון, אתה יכול להגדיר כל אחת ממחלקות אלו בקובץ קוד מקור משלה. למתכנת שמשתמש במחלקותיך יש גישה לכל המחלקות בתוך מרחב השמות באמצעות מילת המפתח using.

כאשר הדבר ניתן ליישום, רצוי להעניק למרחב השמות ברמה העליונה את שם החברה כדי להבטיח ייחודיות. עיין בפרק 3 לקבלת מידע נוסף על קווים מנחים בקביעת שמות.

## מילת המפתח using

לעיתים אתה רוצה לציין סוג נתון בשמו המשויך (qualified) מהצורה namespace.type, אולם הדבר עשוי להיות די מייגע, ולפעמים גם מיותר. בדוגמה הבאה נשתמש באובייקט Console הנמצא במרחב השמות System.

```
class Using1
{
   public static void Main()
   {
      System.Console.WriteLine("test");
   }
}
```

אולם, אם אנו יודעים שהאובייקט Console נמצא רק במרחב השמות System, נוכל לחסוך לעצמנו הקלדה מיותרת. מילת המפתח using מאפשרת להגדיר סדר חיפוש במרחבי שמות, כך שכאשר המהדר פוגש בסוג שאינו משויך, הוא יכול לעבור על רשימת מרחבי השמות המפורטת ולחפש אותו. בדוגמה הבאה, המהדר מאתר את האובייקט Console במרחב השמות System מבלי שהמפתח נדרש להגדירו מחדש בכל פעם.

```
using System;

class Using2
{
   public static void Main()
   {
      Console.WriteLine("test");
   }
}
```

כאשר תתחיל לבנות יישומים של ממש, שיש בהם מאות קריאות לאובייקטים שונים של System, תבין את היתרון בכך שלא צריך לצרף בתחילת כל אחת מההתייחסויות לאובייקטים אלה את השם של מרחב השמות.

אינך יכול להגדיר שם של מחלקה עם מילת המפתח using. לכן, הקוד הבא אינו חוקי:

```
using System.Console; // Invalid.

class Using3
{
   public static void Main()
   {
      WriteLine("test");
   }
}
```

אף על פי כן, באפשרותך להשתמש במתכונת שונה של מילת המפתח using, שבה היא משמשת ככינוי (alias):

```
using Conout = System.Console;

class Using4
{
   public static void Main()
   {
       Conout.WriteLine("test");
   }
}
```

דרך זו מקילה בעת כתיבת קוד, ובמיוחד במקרים שבהם מרחבי שמות מקוננים משולבים ביחד עם שמות מחלקה ארוכים.

# יתרונות CTS

אחת מתכונות המפתח של כל שפה או סביבת ריצה היא יכולת התמיכה שלה בסוגי נתונים. שפה המאפשרת זמינות למספר מוגבל של סוגים, או שמגבילה את יכולת המתכנת להרחיב את הסוגים המובנים בתוכה, אינה בעלת תוחלת חיים ארוכה. מנגד, למערכת סוגים אחידה יש יתרונות רבים אחרים גם כן.

## תפעוליות שיתופית בין שפות

כזכור לך, CLS מגדירה קבוצת חוקים עבור כל מהדר NET ומבטיחה שהקוד הנוצר יפעל באופן שיתופי עם כל יישום NET אחר בסביבת CLR. CTS עצמה מוגדרת ב-CLS (Common Language Specifications) ויש לה חלק מרכזי ביכולת התפעוליות השיתופית

(interoperability) בין שפות, כי היא מגדירה קבוצה של סוגי נתונים שמהדר NET חייב לתמוך בהם בכדי לאפשר שיתוף עם שפות אחרות. לכן, מכיון שכל שפות NET משתמשות במערכת סוגים יחידה, הן יכולות לפעול בשיתוף בצורה חלקה. הצירוף של CTS/CLS הוא זה אשר עוזר לממש את החלום של יכולת התפעוליות השיתופית בין שפות.

## היררכיה המבוססת על מחלקה בודדת

סביבת NET מיישמת את המודל של מחלקת בסיס יחידה, להבדיל מ-++C שבה אין מחלקת בסיס משתמעת עבור כל המחלקות. כל סוג ב-CTS נגזר ממחלקת הבסיס System.Object. גישה זו נתמכת על ידי אנשי התיאוריה של OOP ומהווה את המגמה השלטת כיום בשפות מוכוונות-עצמים. היתרונות בהיררכיה מסוג זה אינם ניכרים בטווח המיידי, אך במהלך הזמן תמצא את עצמך תוהה איך תכננו שפות ללא גישה זו.

היררכיה בעלת מקור בודד היא המפתח למערכת סוגים אחידה, כי היא מבטיחה שכל אובייקט בהיררכיה הוא בעל ממשק משותף. כך, בסופו של דבר, הכל בהיררכיה יהיה מאותו סוג בסיסי (base type). אחד החסרונות העיקריים של ++C הוא חוסר התמיכה בהיררכיה כזו. הדוגמה הבאה תמחיש זאת.

נניח שאתה בונה היררכיה של אובייקטים ב-++C המבוססת על מחלקת בסיס משלך בשם Cfoo. כעת נניח שאתה רוצה למזג אותה עם היררכיה אחרת של אובייקטים, שבה כל האובייקטים נגזרים ממחלקת בסיס בשם Cbar. בדוגמה זו, להיררכיות האובייקטים אין ממשקים תואמים ויידרש מאמץ רב כדי לשלב אותן יחד. אולי יהיה עליך להשתמש בסוג מסוים של מחלקת עטיפה עם צבירה, או אולי תשתמש בהורשה מרובה. עם היררכיה המושתתת על מחלקה בודדת לא קיימת בעיה של חוסר תאימות, מכיון שלכל אובייקט יש אותו הממשק (העובר בהורשה מ- System.Object). כתוצאה מכך, אתה יודע שלכל אחד מהאובייקטים בהיררכיה שלך – ובאופן מכריע, בהיררכיות קוד NET של צד שלישי – יש קבוצה מינימלית מסוימת של פונקציונליות.

## Type Safety

יתרון נוסף ל-CTS הוא הבטחת סוג (type safety). הבטחת סוג דואגת לכך שסוגי נתונים הינם באמת מה שהם אמורים להיות, ושרק פעולות מתאימות יכולות להתבצע על סוג מסוים.

הבטחת סוג מספקת מספר יתרונות ויכולות, כמוסבר להלן, אשר רובם נובעים מהיררכיית אובייקטים בעלת מקור יחיד.

- כל ייחוס לאובייקט הוא מסווג, והאובייקט שאליו הוא מייחס מסווג גם כן. CTS מבטיחה שייחוס מצביע תמיד לאובייקט המשתמע שהוא מצביע אליו.
- מכיון ש-CTS עוקבת אחר כל סוג במערכת, אין דרך להערים על המערכת, כך שתחשוב שסוג אחד הוא למעשה סוג אחר. זהו מקור עיקרי לדאגה בפיתוח יישומים מבוזרים שבהם יש עדיפות לנושא האבטחה.
- כל סוג אחראי להגדרת הגישה אל החברים שלו באמצעות החלת מציין גישה לכל חבר. רמות הגישה כוללות נגישות מלאה לחבר (הצהרה על חבר בתור public), נגישות מוגבלת לחבר רק מתוך מחלקות נגזרות (הצהרה על חבר בתור protected), אי-מתן גישה כלל (הצהרה על חבר בתור private), ואפשרות גישה לחבר אל סוגים אחרים ברכיב הנוכחי בלבד (הצהרה על חבר בתור internal). בפרק הבא נדון במצייני גישה ביתר פירוט.

# סיכום

מערכת הסוגים CTS היא תכונה חשובה של סביבת NET. CTS מגדירה את חוקי מערכת הסוגים שיישומים נדרשים לציית להם בכדי להתבצע באופן תקין ב-CLR. סוגי CTS מתחלקים לשתי קטגוריות: reference types ו-value types. מרחבי שמות יכולים לשמש להגדרת טווח הכרה ביישום. בין היתרונות של מערכת סוגים אוניברסלית נכללים יכולת תפעוליות שיתופית בין שפות, היררכיה הבנויה על מחלקה בודדת והבטחת סוג. ניתן להמיר סוגים ב-C# באמצעות אריזה ופריקה, וניתן ליצור סוגים תואמים השותפים למאפיינים ופונקציונליות באמצעות המרה.

# פרק 5 מחלקות - Classes

מחלקות הן הלב של כל שפה מוכוונת-עצמים. כפי שמתואר בפרק 1 העוסק ביסודות של תכנות מוכוון-עצמים, מחלקה היא כימוס (encapsulation) של נתונים ושל שיטות הפועלות על נתונים אלה. הגדרה זו נכונה בכל שפה מוכוונת-עצמים. מעבר לכך, נבדלות השפות בסוגי הנתונים שניתן לשמור כחברים (as members) וביכולות של כל סוג מחלקה. #C לקחה מעט מהשפות ++C ו-Java בכל הקשור למחלקות ולתכונות רבות, והוסיפה מידה מסוימת של תחכום ליצירת פתרונות אלגנטיים לבעיות ישנות.

בתחילה נתאר את היסודות של הגדרת המחלקות ב-#C, וביניהם חברי מופע (instance members), מצייני גישה (access modifiers), בנאים (constructors), ורשימות אתחול (initialization lists). בהמשך נעבור להגדרת חברים סטטיים (static members) ונסביר את ההבדל שבין קבועים ושדות לקריאה בלבד. לאחר מכן נעסוק במפרקים (destructors) ובדבר הנקרא ניקוי דטרמיניסטי (deterministic finalization). לסיום הפרק נדון בקצרה בהורשה ומחלקות #C.

## הגדרת מחלקות

התחביר המשמש להגדרת מחלקות ב-#C הוא פשוט, במיוחד אם אתה רגיל לכתוב תוכניות ב-++C או Java. אתה מציב את מילת המפתח class לפני שם המחלקה ורושם את חברי המחלקה בין סוגריים מסולסלים:

```
class Employee
{
   private long employeeId;
}
```

זהו הקוד הבסיסי ביותר של מחלקה. המחלקה נקראת Employee ומכילה חבר יחיד בשם employeeId. מילת המפתח private לפני שם החבר נקראת **מציין גישה** (access modifier). #C מגדירה ארבעה מצייני גישה חוקיים שנסקור בהמשך.

# Class Members

בפרק 4 עסקנו בסוגים השונים המוגדרים על ידי מערכת הסוגים CTS. סוגים אלה נתמכים כחברים (members) של מחלקות #C וכוללים:

- **שדות** (fields) – שדה הוא משתנה חבר המשמש לאחסון ערך. בלשון המדוברת של OOP מתייחסים לשדות לעיתים כאל נתוני האובייקט. ניתן להחיל מספר מציינים לשדה הקובעים את אופן השימוש בו. מציינים אלה כוללים static, readonly ו-const. בקרוב נדון במשמעותם ובאופן השימוש בהם.

- **שיטות** (methods) – שיטה היא קוד ממשי הפועל על נתוני האובייקט (או שדות). בפרק זה נתמקד בהגדרת נתוני המחלקה. בפרק 6 נעסוק בשיטות ביתר פירוט. ראוי לציין שיש הנוהגים לקרוא ל-method בשם "פונקציה".

- **מאפיינים** (properties) – מאפיינים נקראים לעיתים **שדות מתוחכמים** (smart fields), מכיון שלמעשה אלו הן שיטות הנראות כשדות ללקוחות המחלקה. הדבר מאפשר רמה גבוהה יותר של הפשטה עבור הלקוח, והוא אינו צריך לדעת אם עליו לגשת ישירות לשדה או באמצעות שיטת גישה. בפרק 7 נעסוק במאפיינים ביתר פירוט.

- **קבועים** (constants) – קבוע הוא שדה עם ערך, שאינו ניתן לשינוי. יותר מאוחר בפרק זה נעסוק בקבועים ונשווה אותם לשדות לקריאה בלבד.

- **סדרנים** (indexers) – באותה מידה שמאפיין הוא שדה מתוחכם (smart field), סדרן הוא מערך מתוחכם (smart array). כלומר, זהו חבר המאפשר גישה קלה לאובייקט כאילו היה מערך, למטרת קריאה או הגדרה של ערכים באמצעות שיטות הגישה get ו-set. סדרנים ומאפיינים נסקרים בפרק 7.

- **אירועים** (events) – אירוע הוא מה שגורם לקטע קוד מסוים להתבצע. אירועים הם חלק בלתי נפרד מתכנות במערכת Microsoft Windows. אירוע נוצר, לדוגמה, כאשר מזיזים את העכבר, לוחצים על חלון או משנים את גודלו. האירועים ב-#C משתמשים בתבנית התכנון הסטנדרטית publish/subscribe שקיימת ב-MSMQ (Microsoft Message Queuing) ובמודל האירועים האסינכרוני של COM+. היא מעניקה ליישום יכולת לטפל באירועים באופן אסינכרוני – אך ב-#C תבנית תכנון זו היא מושג של "מחלקה ראשונה" ("first-class") המובנה בתוך השפה. פרק 14 מסביר איך להשתמש באירועים.

- **אופרטורים** (operators) – #C מאפשרת להוסיף אופרטורים מתמטיים סטנדרטיים למחלקה בדרך של העמסת אופרטורים (operator overloading). כך ניתן לכתוב באמצעותם קוד אינטואיטיבי יותר. העמסת אופרטורים נסקרת בפרק 13.

# Access Modifiers

לאחר שהכרנו את הסוגים השונים שניתן להגדיר כחברים במחלקה של #C, נתייחס למציין חשוב המאפשר להגדיר באיזו מידה חבר מחלקה (class member) נתון הוא זמין או נגיש לקוד מחוץ למחלקה שלו. מציינים אלה נקראים מצייני גישה (access modifiers) ומפורטים בטבלה 5-1.

**טבלה 5-1** מצייני גישה של #C

| מציין גישה | תיאור |
|---|---|
| public | ציבורי – החבר נגיש לקוד הנמצא מחוץ להכרזת המחלקה וההיררכיה של מחלקות נגזרות. |
| protected | מוגן – החבר נגיש למחלקה עצמה ולמחלקות נגזרות בלבד. |
| private | פרטי – החבר נגיש למחלקה עצמה בלבד. לפיכך, אפילו מחלקות נגזרות אינן יכולות לגשת לחברים אלה. |
| internal | פנימי – החבר נגיש ליחידת ההידור הנוכחית בלבד. מציין גישה זה יוצר hybrid (בן-כלאיים) של נגישות ציבורית ומוגנת כתלות במיקומו של הקוד. |

שים לב שמציין הגישה של ברירת המחדל לחבר הוא private. לכן, אם אתה רוצה מציין גישה אחר עבורו, עליך להחילו באופן מפורש. זאת, בניגוד ל-++C שבה חבר שאינו מוגדר עם מציין גישה מפורש מקבל את תכונות הנגישות של מציין הגישה האחרון שהוכרז. לדוגמה, בקוד ++C הבא החברים a, b ו-c מוגדרים עם נגישות של public, והחברים d ו-e מוגדרים כ-protected:

```
class CAccessModsInCpp
{
   public:
   int a;
   int b;
   int c;
   protected:
      int d;
      int e;
}
```

כדי להשיג יעד זהה ב-#C יש לשנות את הקוד, כך:

```
class AccessModsInCSharp
{
   public int a;
   public int b;
   public int c;
   protected int d;
   protected int e;
}
```

בקוד #C הבא יוכרז החבר b כ-private:

```
public MoreAccessModsInCSharp
{
   public int a;
   int b;
}
```

# השיטה Main

כל יישום #C חייב להכיל שיטה Main המוגדרת באחת ממחלקותיו. בנוסף לכך, השיטה חייבת להיות מוגדרת כ-public ו-static (במשמעות של static נדון בהמשך). בניגוד ל- ++C, מהדר #C אינו מייחס חשיבות למיקום של השיטה Main ולכן תוכל לבחור בכל מחלקה למטרה זו, מבלי שהדבר ישפיע על סדר ההידור. המהדר מספיק מתוחכם כדי לסקור את קבצי המקור ולאתר את השיטה Main בכוחות עצמו. עם זאת, יש לזכור ששיטה חשובה זו היא **נקודת הכניסה** בכל יישום של #C.

למרות שתוכל להציב את השיטה Main בכל מחלקה, אני ממליץ ליצור עבורה מחלקה מיוחדת. הנה דוגמה לכך, עם שימוש במחלקה הפשוטה Employee:

```
class Employee
{
   private int employeeId;
}

class AppClass
{

   static public void Main()
   {
      Employee emp = new Employee();
   }
}
```

דוגמה בסיסית זו כוללת שתי מחלקות. היא מייצגת את הגישה המקובלת לתכנות ב-C#, ואפילו ביישומים הפשוטים ביותר. המחלקה הראשונה (Employee) היא למעשה מרחב הבעיה, והמחלקה השנייה (AppClass) מכילה את נקודת הכניסה הדרושה ליישום. במקרה זה, השיטה Main יוצרת מופע של אובייקט Employee, ואם היה זה יישום מציאותי, הוא היה גם עושה משהו עם החברים של האובייקט Employee.

## ארגומנטים של שורת-פקודה

ניתן לספק ליישום ארגומנטים של שורת-פקודה על ידי הכרזה על השיטה Main כמקבלת ארגומנט יחיד של מערך מסוג string. בנקודה זו ניתן לעבד את הארגומנטים כפי שעושים בכל מערך. אף על פי שמערכים לא ייסקרו עד לפרק 7, הנה קוד כללי לביצוע איטרציה על הארגומנטים של שורת-פקודה עבור יישום, וכתיבתם להתקן הפלט הסטנדרטי:

```
using System;
class CommandLineApp
{
   public static void Main(string[] args)
   {
      foreach (string arg in args)
      {
         Console.WriteLine("Argument: {0}", arg);
      }
   }
}
```

הנה דוגמה של קריאה ליישום עם מספר ערכים שנבחרו באקראי:

```
e:>CommandLineApp 5 42
Argument: 5
Argument: 42
```

הארגומנטים של שורת-הפקודה ניתנים במתכונת של מערך מחרוזות. בכדי לעבד את הפרמטרים כדגלים (flags) או מתגים (switches) עליך לתכנת יכולת זו בעצמך.

מפתחים ב- Microsoft Visual C++ רגילים לבצע איטרציה על מערך המייצג את הארגומנטים של שורת-הפקודה עבור יישום. עם זאת, בניגוד ל- Visual C++, מערך הארגומנטים ב-C# אינו מכיל את שם היישום ככניסה הראשונה במערך.

## ערכים מוחזרים

רוב הדוגמאות בספר זה מגדירות כך את השיטה Main:

```
class SomeClass
{
.........
public static void Main()
{
........
}
........
}
```

אפשר להגדיר את השיטה Main באופן שתחזיר ערך מסוג int. למרות שאין זה מקובל ביישומי GUI, הרי שזה יכול להיות שימושי בכתיבת יישומי מסוף (console applications) המתוכננים להיות מופעלים באצווה. המשפט return מסיים את ביצוע השיטה והערך המוחזר משמש כדרגת שגיאה (error level) עבור היישום הקורא, או קובץ האצווה, לציון הצלחה או כישלון המוגדרים על ידי המשתמש. לשם כך השתמש באבטיפוס הבא:

```
public static int Main()
{
.........
   // Return some value of type int
   // that represents success or value.
   return 0;
}
```

## הרבה מחלקות עם השיטה Main

מתכנני #C כללו מנגנון שבאמצעותו תוכל להגדיר יותר ממחלקה אחת עם השיטה Main. סיבה אחת שמצדיקה זאת היא הכנסת קוד ניסוי במחלקות. באמצעות המתג /main:<className> תוכל לציין למהדר #C באיזו מבין שיטות Main (ובאיזו מחלקה) ברצונך להשתמש באותה הרצה של היישום. הנה דוגמה הכוללת שתי מחלקות המכילות שיטות Main:

```
using System;

class Main1
{
   public static void Main()
   {
      Console.WriteLine("Main1");
   }
}
```

```
class Main2
{
   public static void Main()
   {
      Console.WriteLine("Main2");
   }
}
```

כדי להדר את היישום, כך שנקודת הכניסה תהיה השיטה Main1.Main, עליך להשתמש במתג בדרך זו:

```
csc MultipleMain.cs /main:Main1
```

אם תשנה את המתג ל- /main:Main2, תהיה נקודת הכניסה בשיטה Main2.Main.

מכיוון ש-#C רגישה לאותיות רישיות, עליך להקפיד להשתמש באותיות הנכונות גם בשם המחלקה המופיע במתג. בנוסף לכך, אם תנסה להדר יישום המורכב ממחלקות מרובות עם שיטות Main מוגדרות, אך לא תציין את המתג /main – תקבל הודעה על שגיאת הידור.

# Constructors

אחד היתרונות הבולטים של שפות OOP כמו #C, הוא ביכולת להגדיר שיטות מיוחדות הנקראות בכל פעם שנוצר מופע של מחלקה. שיטות אלו נקראות **בנאים** (constructors). #C מציגה סוג חדש של בנאי הנקרא **בנאי סטטי** (static constructor), שאותו תכיר בסעיף הבא.

יתרונו העיקרי של הבנאי נובע מכך שהוא מאתחל את האובייקט בצורה נכונה לפני השימוש בו. כאשר המשתמש יוצר מופע של אובייקט, נקרא הבנאי של האובייקט והוא חייב לחזור בטרם אפשר יהיה להשתמש באובייקט זה. זו הערובה שמסייעת להבטיח את שלמות האובייקט ועוזרת ליצור יישומים הרבה יותר מהימנים כאשר הם נכתבים בשפות מוכוונות-עצמים.

לבנאי צריך לקבוע שם, כדי שהמהדר ידע לקרוא לו ביצירת המופע. מתכנני #C הלכו בעקבות Stroustrup (יוצר שפת ++C) וקבעו שהבנאים ב-#C חייבים להיות בעלי אותו שם כמו של המחלקה עצמה.

הנה מחלקה פשוטה, עם בנאי פשוט באותה מידה:

```
using System;

class Constructor1App
{
   Constructor1App()
   {
      Console.WriteLine("I'm the constructor.");
   }

   public static void Main()
   {
      Constructor1App app = new Constructor1App();
   }
}
```

בנאים אינם מחזירים ערכים. אם תנסה להקדים לשם הבנאי סוג כלשהו, המהדר יציג הודעת שגיאה האומרת שאינך יכול להגדיר חבר באותו השם כמו הסוג המכיל אותו.

עליך גם לשים לב לדרך יצירת המופע מאובייקטים ב-C#. הדבר נעשה באמצעות מילת המפתח **new** בתחביר הבא:

```
<class> <object> = new <class> (constructor arguments)
```

אם יש לך רקע עבודה ב-C++, שים לב לכך באופן מיוחד. ב-C++ אתה יכול ליצור מופע של אובייקט בשתי דרכים. אתה יכול להכריז עליו במחסנית באופן הבא:

```
// C++ code. This creates an instance of CMyClass on the stack.
CMyClass myClass;
```

לחילופין, אתה יכול ליצור מופע של האובייקט בזיכרון חופשי (heap) באמצעות מילת המפתח New של C++:

```
// C++ code. This creates an instance of CMyClass on the heap.
CMyClass myClass = new CMyClass();
```

יצירת מופע מאובייקטים ב-C# נעשה בדרך שונה ומהווה מקור לבלבול עבור מפתחים חדשים בשפת C#. הבלבול נובע מהעובדה שמילת המפתח ליצירת אובייקטים משותפת לשתי השפות. אף על פי שמילת המפתח new ב-C++ מאפשרת להכתיב היכן נוצר האובייקט, ב-C# זה תלוי בסוג שמממשים. כפי שראינו בפרק 4, סוגי ייחוס (reference types) נוצרים ב-heap וסוגי ערכים (value types) נוצרים במחסנית. לפיכך, מילת המפתח new מאפשרת ליצור מופע חדש של אובייקט, אך אינה קובעת היכן הוא נוצר.

מכאן, הקוד הבא הוא חוקי ב-#C, אך אם אתה מפתח ב-++C קוד זה אינו מבצע את מה שאולי אתה חושב שהוא עושה:

```
MyClass myClass;
```

ב-++C קוד זה יוצר מופע של MyClass במחסנית. כפי שהוזכר, אתה יכול ליצור אובייקטים ב-#C רק על ידי שימוש במילת המפתח new. לכן, שורת קוד זו ב-#C מצהירה למעשה ש-myClass הוא משתנה מסוג MyClass, אך אינה יוצרת מופע של האובייקט.

כדוגמה לכך, אם אתה מהדר את התוכנית הבאה, מהדר #C יזהיר אותך שהמשתנה הוכרז, אך לעולם לא ייעשה בו שימוש ביישום:

```
using System;

class Constructor2App
{
   Constructor2App()
   {
      Console.WriteLine("I'm the constructor");
   }

   public static void Main()
   {
      Constructor2App app;
   }
}
```

לכן, אם אתה מכריז על סוג אובייקט, עליך לממש אותו במקום כלשהו בקוד באמצעות מילת המפתח new:

```
Constructor2App app;
app = new Constructor2App();
```

מדוע שתכריז על אובייקט בלי ליצור מופע שלו? הכרזה על אובייקטים לפני השימוש בהם נעשית במקרים שבהם אתה מכריז על מחלקה אחת בתוך מחלקה אחרת. קינון מחלקות בדרך זו נקרא containment (הכלה), או aggregation.

## Instance Members-ו Static Members

כמו ב- ++C, ניתן להגדיר חבר מחלקה כ**חבר סטטי** (static member) או **חבר מופע** (instance member). כברירת מחדל, כל חבר מוגדר כחבר מופע, כלומר עותק של חבר נוצר עבור כל מופע של המחלקה. כאשר מכריזים על חבר סטטי, יש עותק אחד בלבד של החבר.

חבר מחלקה סטטי נוצר כאשר יישום המכיל את המחלקה נטען, והוא נשאר קיים במשך כל חיי היישום. לפיכך אתה יכול לגשת לחבר סטטי (static member) גם אם טרם נוצר מופע של המחלקה. מדוע תרצה לעשות זאת?

דוגמה אחת כרוכה בשיטה Main. סביבת הריצה CLR זקוקה לנקודת כניסה משותפת אל היישום. כדי שלא יהיה צורך לממש אחד מהאובייקטים, הכלל הוא להגדיר שיטה סטטית הנקראת Main באחת מהמחלקות. תרצה להשתמש בחברים סטטיים גם כאשר שיטה שייכת למחלקה מבחינה סמנטית, אך אינה זקוקה לאובייקט ממשי.

לדוגמה, כשאתה רוצה לעקוב אחר מספר המופעים של אובייקט נתון שנוצרו במשך חיי היישום. הקוד הבא תקין, מכיון שחברים סטטיים חיים מעבר למופעים של אובייקט:

```
using System;

class InstCount
{
   public InstCount()
   {
   instanceCount++;
   }
    static public int instanceCount = 0;
}

class AppClass
{
   public static void Main()
   {
      Console.WriteLine(InstCount.instanceCount);

      InstCount ic1 = new InstCount();
      Console.WriteLine(InstCount.instanceCount);

      InstCount ic2 = new InstCount();
      Console.WriteLine(InstCount.instanceCount);
   }
}
```

דוגמה זו מפיקה את הפלט הבא:

```
0
1
2
```

הערה בהקשר של חברים סטטיים: חבר סטטי חייב להיות בעל ערך חוקי. תוכל להגדיר ערך זה בזמן שאתה מגדיר את החבר, הנה כך:

```
static public int instanceCount1 = 10;
```

אם אינך מאתחל את המשתנה, CLR יעשה זאת בעת הפעלת היישום על ידי שימוש בערך מחדל של 0. לכן, שתי השורות הבאות שקולות:

```
static public int instanceCount2;
static public int instanceCount2 = 0;
```

## Constructor Initializers

כל בנאי האובייקטים ב-#C – חוץ מהבנאים של System.Object – כוללים קריאה לבנאי של מחלקת הבסיס מייד לפני ביצוע שורת הקוד הראשונה של הבנאי. **מאתחלי הבנאים** (constructor initializers) מאפשרים להגדיר לאיזו מחלקה ולאיזה בנאי אתה רוצה לקרוא. יש לכך שתי צורות:

- מאתחל מהצורה **base(...)** מאפשר קריאה לבנאי מחלקת הבסיס של המחלקה הנוכחית, כלומר קריאה לבנאי המסוים המשתמע מצורת הבנאי שנקרא.
- מאתחל מהצורה **this(...)** מאפשר למחלקה הנוכחית לקרוא לבנאי אחר המוגדר בתוכה. הדבר שימושי כאשר העמסת בנאים מרובים ואתה רוצה להבטיח שתמיד תהיה קריאה לבנאי של ברירת המחדל. שיטות מועמסות נסקרות בפרק 6, אך הנה הגדרה מהירה וגסה: שיטות מועמסות הן שתי שיטות או יותר, בעלות אותו שם אך בעלות רשימות ארגומנטים שונות.

בדוגמה הבאה, הקוד יבצע את הבנאי למחלקה A תחילה ואז את הבנאי למחלקה B:

```
using System;

class A
{
   public A()
   {
      Console.WriteLine("A");
   }
}

class B : A
```

```
{
   public B()
   {
      Console.WriteLine("B");
   }
}

class DefaultInitializerApp
{
   public static void Main()
   {
      B b = new B();
   }
}
```

קוד זה שקול לקוד הבא, שבו בנאי מחלקת הבסיס נקרא באופן מפורש:

```
using System;

class A
{
   public A()
   {
      Console.WriteLine("A");
   }
}

class B : A
{
   public B() : base()
   {
      Console.WriteLine("B");
   }
}

class BaseDefaultInitializerApp
{
   public static void Main()
   {
      B b = new B();
   }
}
```

הבה נתבונן בדוגמה הממחישה טוב יותר מתי מאתחלים של בנאים הם שימושיים. גם בדוגמה זו לפנינו שתי מחלקות: A ו-B. כעת, מחלקה A מכילה שני בנאים, אחד ללא ארגומנטים ואחד המקבל ארגומנט מסוג int. מחלקה B מכילה בנאי אחד המקבל int.

הבעיה מתעוררת בבנאי של מחלקה B. כאשר נפעיל את הקוד הבא, ייקרא הבנאי ללא הארגומנטים:

```
using System;

class A
{
   public A()
  {
      Console.WriteLine("A");
   }

   public A(int foo)
   {
      Console.WriteLine("A = {0}", foo);
   }
}

class B : A
{
   public B(int foo)
  {
      Console.WriteLine("B = {0}", foo);
   }
}

class DerivedInitializer1App
{
   public static void Main()
   {
      B b = new B(42);
   }
}
```

ובכן, איך ניתן להבטיח קריאה לבנאי הרצוי במחלקה A? על ידי הוראה מפורשת למהדר, ברשימת האתחול, איזה בנאי עליו לקרוא. הנה דוגמה:

```
using System;

class A
{
   public A()
   {
      Console.WriteLine("A");
   }
```

```
    public A(int foo)
    {
        Console.WriteLine("A = {0}", foo);
    }
}

class B : A
{
    public B(int foo) : base(foo)
    {
        Console.WriteLine("B = {0}", foo);
    }
}

class DerivedInitializer2App
{
    public static void Main()
    {
        B b = new B(42);
    }
}
```

שלא כמו ב- Visual C++, אינך יכול להשתמש במאתחלי בנאי בכדי לגשת לחברי מופע של המחלקה הנוכחית, מלבד בנאים.

# Constants vs. Read-Only Fields

לעיתים תרצה שדות שאינך רוצה לשנות במהלך ביצוע היישום – לדוגמה, קבצי נתונים שהיישום תלוי בהם, הערך pi עבור מחלקת מתמטיקה, או כל ערך אחר שבו אתה משתמש ביישום ויודע שלא ישתנה לעולם. בכדי להתמודד עם מצבים אלה מאפשרת C# להגדיר שני סוגי חברים קרובים: **קבועים** (constants) **ושדות לקריאה בלבד** (read-only fields).

## Constants

כפי שתוכל לנחש לפי השם, קבועים, המיוצגים באמצעות מילת המפתח **const**, הם שדות הנשארים קבועים במשך כל חיי היישום. יש לזכור שני חוקים בלבד כאשר מגדירים משהו כ-const. ראשית, קבוע הוא חבר מחלקה (member) שערכו נקבע בזמן ההידור על ידי המתכנת, או על ידי המהדר, עם ערך ברירת מחדל. שנית, את ערכו של חבר קבוע יש לקבוע כפשוטו, כלומר לכתוב את הערך עצמו בקוד.

בכדי להגדיר חבר כקבוע, יש לציין את מילת המפתח const לפני החבר, כך:

```
using System;

class MagicNumbers
{
   public const double pi = 3.1415;
   public const int answerToAllLifesQuestions = 42;
}

class ConstApp
{
   public static void Main()
   {
      Console.WriteLine("pi = {0}, everything else = {1}",
         MagicNumbers.pi, MagicNumbers.answerToAllLifesQuestions);
   }
}
```

שים לב לנקודה חשובה: הלקוח אינו צריך לממש את המחלקה MagicNumbers מכיון שחברים קבועים הם סטטיים כברירת מחדל. כדי לקבל תמונה טובה יותר, התבונן בקוד MSIL הבא שנוצר עבור שני חברים אלה:

```
answerToAllLifesQuestions : public static literal int32 = int32(0x0000002A)
pi : public static literal float64 = float64(3.1415000000000002)
```

## Read-Only Fields

שדה המוגדר כ-const הוא שימושי, מכיון שהוא מתעד בצורה ברורה את כוונת המתכנת להכיל בשדה ערך שאינו משתנה. עם זאת, הדבר טוב רק אם אתה יודע את הערך בזמן ההידור. מה עושה מתכנת כאשר מתעורר הצורך בשדה עם ערך שאינו ידוע עד לזמן הריצה ואינו צריך להשתנות מרגע שאותחל? סוגיה זו – אשר בדרך כלל אין מתמודדים איתה בשפות אחרות – נפתרה על ידי המתכננים של שפת C# בעזרת "שדה לקריאה בלבד" (read-only field).

שדה מוגדר לקריאה בלבד עם מילת המפתח **readonly** והוא יכול לקבל ערך במקום אחד בלבד: הבנאי. מנקודה זו ואילך, גם המחלקה עצמה וגם לקוחות המחלקה אינם יכולים לשנות את ערכו של השדה. נניח שאתה רוצה לעקוב אחר הרזולוציה של התצוגה עבור יישום גרפיקה. אינך יכול לטפל בבעיה זו עם const, מכיון שהיישום אינו יכול לקבוע את הרזולוציה של משתמש הקצה עד לזמן הריצה.

לכן תכתוב קוד כמו זה:

```
using System;

class GraphicsPackage
{
   public readonly int ScreenWidth;
   public readonly int ScreenHeight;

   public GraphicsPackage()
   {
      this.ScreenWidth = 1024;
      this.ScreenHeight = 768;
   }
}

class ReadOnlyApp
{
   public static void Main()
   {
      GraphicsPackage graphics = new GraphicsPackage();
      Console.WriteLine("Width = {0}, Height = {1}",
                        graphics.ScreenWidth,
                        graphics.ScreenHeight);
   }
}
```

ממבט ראשון נראה כי קוד זה מתאים לדרישתך. עם זאת, מתעורר כאן עניין קטן: השדות לקריאה בלבד שהגדרנו הם שדות מופע. המשמעות היא שהמשתמש יידרש לממש את המחלקה כדי להשתמש בשדות. לא תהיה בעיה לקבוע את ערכי השדות האלה במקרים שבהם המחלקה ממומשת. אך מה אם אתה רוצה קבוע שהוא סטטי על פי ההגדרה, שניתן לאתחל אותו בזמן ריצה? במקרה זה, תגדיר את השדה עם המציינים static ו-readonly גם יחד. לאחר מכן תיצור סוג מיוחד של בנאי הנקרא **בנאי סטטי** (static constructor). בנאים סטטיים משמשים לאתחול שדות סטטיים, לקריאה בלבד או לפעולה אחרת. כאן שיניתי את הדוגמה הקודמת, כך ששדות הרזולוציה יהיו סטטיים ולקריאה בלבד, והוספתי בנאי סטטי. שים לב לתוספת מילת המפתח static בהגדרת הבנאי:

```
using System;

class GraphicsPackage
{
   public static readonly int ScreenWidth;
   public static readonly int ScreenHeight;
```

```
    static GraphicsPackage()
    {
        // Code would be here to
        // calculate resolution.
        ScreenWidth = 1024;
        ScreenHeight = 768;
    }
}

class ReadOnlyApp
{
    public static void Main()
    {
        Console.WriteLine("Width = {0}, Height = {1}",
                          GraphicsPackage.ScreenWidth,
                          GraphicsPackage.ScreenHeight);
    }
}
```

## Garbage Collector

אחת התכונות החשובות ביותר של מערכת מבוססת רכיבים היא יכולתה לספק ניקוי ושחרור של משאבים כאשר רכיבים מסיימים לפעול. ב"ניקוי ושחרור משאבים" כוונתי לשחרור בעיתוי נכון של ייחוסים לרכיבים אחרים, וגם שחרור של משאבים יקרים או מוגבלים שיש להם ביקוש, כגון חיבורים למסדי נתונים, ידיות לקבצים ויציאות תקשורת. ב"כאשר רכיבים מסיימים לפעול" כוונתי לנקודה שבה האובייקט אינו בשימוש יותר.

בשפת C++ המושג "ניקוי" הוא פשוט וברור, מכיון שהוא מתבצע במפרק (destructor) של האובייקט, פונקציה המוגדרת עבור כל אובייקט C++ ומתבצעת באופן אוטומטי כאשר האובייקט יוצא מטווח ההכרה. בעולם של NET, ניקוי אובייקט מטופל באופן אוטומטי על ידי Garbage Collector (אוסף האשפה – GC) של NET. אסטרטגיה זו שנויה במחלוקת, כי בניגוד לאפשרות החיזוי ב-C++ מתי מתבצע קוד הסיום של אובייקט, הפתרון של NET מבוסס על מודל "עצל".

Garbage Collector מפעיל מטלות (threads) שמתבצעות ברקע וקובעות מתי אין יותר התייחסות לאובייקט מסוים. רק אז אחראיות מטלות GC אחרות להרצת קוד הסיום של אותו אובייקט. פתרון זה פועל בצורה נאה ברוב המקרים אך הוא פחות מאופטימלי כאשר מטפלים במשאבים שצריך לשחררם בזמן רצוי ובסדר הניתן לחיזוי. כפי שאולי אתה חש, אין זו בעיה קלה לפתרון.

# מעט היסטוריה

לפני מספר שנים, כאשר החל פרויקט NET, התעורר ויכוח רציני בסוגיית ניהול משאבים. התורמים הראשונים ל-NET הגיעו מהצוותים של COM ו- Visual Basic, וגם מצוותים רבים אחרים ממחלקות פיתוח שונות של מיקרוסופט. אחת הבעיות הגדולות שניצבה בפני צוותים אלה היתה, איך לטפל בסוגיות הקשורות לספירת ייחוסים (references), כולל ייחוסים מעגליים ושגיאות הנובעות משימוש לא נכון בהם. דוגמה אחת לכך היא בעיית הייחוס המעגלי: אובייקט אחד מכיל ייחוס לאובייקט אחר אשר לאחר מכן מכיל ייחוס חזרה לאובייקט הראשון. מתי ואיך שחרור של ייחוס מעגלי כזה יכול לגרום לבעיות? אי שחרור של אחד הייחוסים או שניהם גורם לדליפת זיכרון שקשה מאוד לעלות על עקבותיה. כל מי שמנוסה ב-COM מכיר את הסיפורים על זמן שהולך לאיבוד במרדף אחר באגים הנובעים מבעיות של ספירת ייחוסים. מיקרוסופט הבינה שבעיות של ספירת ייחוסים הן שכיחות למדי ביישומים מבוססי-רכיבים חשובים (כמו יישומי COM) ונערכה לספק פתרון אחיד ב-NET.

הפתרון הראשון לבעיה היה מבוסס על ספירת ייחוסים אוטומטית, כך שאם המתכנת שכח לספור ייחוס לאחד מממשקי האובייקט הדבר לא ישפיע כלל. זאת, בנוסף לשיטות לאיתור וטיפול בייחוסים מעגליים באופן אוטומטי. בנוסף לכך, צוותי NET בחנו את האפשרות להוסיף אוסף שמרני של ייחוסים (reference collection), אוסף מעקב (tracing collection) ואלגוריתמים של GC שיוכלו לאסוף אובייקט בודד ללא צורך במעקב מלא על גרף האובייקט. עם זאת, בגלל מיגוון סיבות (מייד אסקור אותן) הסיקו שפתרונות אלה לא יתאימו עבור המקרה הרגיל.

מכשול אחד עיקרי נובע מכך שבימים הראשונים לפיתוחה של NET, היעד הראשון היה לקיים מידה גדולה של תאימות עם Visual Basic. לכן, הפתרון היה צריך להיות מושלם ושקוף ללא שינויים סמנטיים בשפת Visual Basic עצמה. לאחר ויכוח ממושך, הפתרון הסופי היה מודל מבוסס-הקשר (context-based model), שבו כל דבר הקיים בהקשר דטרמיניסטי (שניתן לקבוע אותו מראש) ישתמש בספירת ייחוסים (מעל ה-GC), וכל דבר שאינו כלול בהקשר זה ישתמש רק ב-GC. זה אמנם מנע חלק מהסוגיות של הסתעפות סוגים (המתוארות בהמשך), אך לא הניב פתרון טוב לעירוב פשוט של קוד בין שפות. כך הוחלט לבצע סדרה של שינויים בשפת Visual Basic בכדי להפוך אותה ל"מודרנית" ובעלת יכולות רבות יותר. חלק מההחלטה היה להשמיט את דרישות התאימות לאורך החיים ב- Visual Basic. החלטה זו גם סיימה למעשה את החקירה בסוגיות של משך חיים דטרמיניסטי.

## Deterministic Finalization

לאחר היסטוריה קצרה זו, הבה נגדיר מהו **גמר דטרמיניסטי** (deterministic finalization). מרגע שנקבע כי אובייקט אינו בשימוש, מתבצע קוד הסיום שלו ומשחרר את הייחוסים לאובייקטים אחרים שבהם הוא אוחז.

תהליך סיום זה גולש בצורה טבעית במורד גרף האובייקטים, החל מהאובייקט העליון. באופן אידיאלי תהליך זה היה מתפקד בצורה נכונה עבור אובייקטים שיתופיים וגם עבור אובייקטים לשימוש יחיד.

שים לב שהזמן כאן אינו מובטח. מרגע שמטלת GC מגלה שייחוס אינו בשימוש, מטלה מסוימת זו לא תעשה דבר עד שקוד הסיום של האובייקט יתבצע. עם זאת, במהלך העיבוד עשוי תהליך אחר לתפוס את מקומו (context switch). לכן, מנקודת המבט של היישום, עשוי לחלוף זמן לא מוגדר עד לסיומו של שלב זה. כפי שצוין לעיל, קיימים מצבים רבים שבהם התזמון או סדר הביצוע ביחס לקוד הסיום חשוב עבור היישום. מצבים אלה קשורים בדרך כלל למשאבים מבוקשים.

הנה מספר דוגמאות למשאבים שעל האובייקט לשחרר מרגע שאין בו שימוש:

- **Memory** – שחרור זיכרון עבור גרף אובייקט, מחזיר את הזיכרון למאגר וניתן להשתמש בו.
- **Window handles** – טביעת הרגל (footprint) של אובייקט חלון ב-GC אינה משקפת את המחיר בפועל. יש מידה מסוימת של טביעת רגל בתוך מערכת ההפעלה לשם ייצוג החלון, וייתכן שיש גם מגבלה אחרת, מלבד זיכרון פנוי, על מספר ה-handles הכולל של חלונות שניתן להקצות.
- **Database connections** – חיבורים בו-זמנית למסד נתונים נרשמים באופן שוטף, ולכן מספר החיבורים הזמינים עשוי להיות מוגבל. חשוב שחיבורים אלה יוחזרו למאגר הזיכרון במהירות, כדי שניתן יהיה להשתמש בהם שוב.
- **Files** – מכיון שיש מופע יחיד של קובץ נתון ונדרשת גישה בלעדית עבור פעולות רבות, חשוב שה-file handles תיסגרנה כאשר הוא אינו בשימוש.

### Reference-Counting Collection

**ספירת ייחוסים** (reference counting) תורמת במידה ניכרת לגמר דטרמיניסטי ברוב המקרים, אך אין לכך כל ערך אם קיימים מספר מקרים שבהם אינה מועילה, כמו במקרה של ייחוסים מעגליים. למעשה, ספירת ייחוסים פשוטה לעולם אינה אוספת אובייקטים המשתתפים במעגלים. רבים מאיתנו למדו בכאב טכניקות ידניות לטפל בכך, אך השימוש בהם מהווה מקור לשגיאות תוכנה. בעיה נוספת מתעוררת לגבי אובייקטים מרוחקים. יש הטוענים כי מרגע שנותנים ייחוס לאובייקט מחוץ לשליטה המיידית של תוכנית צמודה במהודק, מאבדים את הגמר הדטרמיניסטי, כי לא ידוע מתי או אם הקוד ה"חיצוני" ישחרר את הייחוס. יש הסבורים שמערכת מורכבת התלויה בסדר הסיום של גרף אובייקטים מורכב היא רגישה מטבעה, וסביר שתגרור בעיות תחזוקה משמעותיות ככל שהקוד גדל.

### Tracing Collection

**אוסף מעקב** (tracing collector) מבטיח מעט פחות מספירת ייחוסים. במידת מה, זו מערכת עצלה יותר, בעלת התחשבות בביצוע קוד סיום. לאובייקטים יש finalizers, שיטות המתבצעות כאשר האובייקט אינו ניתן יותר להשגה על ידי התוכנית. היתרון במעקב הוא שייחוסים מעגליים אינם סוגיה שיש להתמודד איתה, ויתרון גדול יותר הוא שהצבת ייחוס היא פעולת העברה פשוטה. המחיר שמשלמים הוא, שאין ערובה לכך שקוד הסיום יתבצע "מיידית" לאחר שהייחוס אינו עוד בשימוש. בתוכניות המתנהגות בצורה נאותה תתבצע קריאה ל-finalizers (מסיימים, גומרים) עבור אובייקטים (תוכנית המתנהגת בצורה מזיקה היא זו שנופלת או מכניסה את מטלת ה-finalizer ללולאה אינסופית). התיעוד המקוון נוטה להיות זהיר מדי בהבטחות, אך אם לאובייקט יש שיטת finalizer המערכת תקרא לה. אין בכך התמודדות עם סוגיית גמר דטרמיניסטי, אך חשוב להבין שמשאבים אכן ייאספו ו-finalizers הם דרך יעילה למנוע דליפת משאבים בתוכנית.

## ביצועים

ביצועים הם סוגיה חשובה ביחס לבעיית הגמר (finalization). צוות NET מאמין שחייב להיות סוג מסוים של אוסף מעקב כדי לטפל בייחוסים מעגליים, אשר דורש חלק גדול מהמחיר של אוסף מעקב. כמו כן הם מאמינים שביצועי קוד עשויים להיות מושפעים באופן ממשי מהמחיר של ספירת ייחוסים. החדשות הטובות הן, שמתוך כל האובייקטים שהוקצו בתוכנית שמתבצעת, מספר האובייקטים שזקוקים באמת לגמר דטרמיניסטי הוא קטן.

עם זאת, קשה בדרך כלל לבודד את עלות הביצועים רק לאובייקטים אלה. הבה נתבונן בפסידו-קוד של הצבת ייחוס פשוטה כאשר משתמשים באוסף מעקב ואז נתבונן בפסידו-קוד כאשר משתמשים בספירת ייחוסים:

```
// Tracing.
a = b;
```

זהו זה. המהדר הופך זאת להוראת העברה (move) יחידה, ואולי אף ממטב (optimize) זאת בנסיבות מסוימות.

```
// Ref counting.
if (a != null)
   if (InterlockedDecrement(ref a.m_ref) == 0)
      a.FinalRelease();

if (b != null)
   InterlockedIncrement(ref b.m_ref);

a = b;
```

זהו קוד נפוח למדי, המגדיל את מערך העבודה ומוריד באופן גס את הביצועים, במיוחד כאשר נתונות שתי ההוראות הנועלות זו את זו (interlock). תוכל להגביל את נפח הקוד על ידי הצבת כל זה בשיטת עזר, ובכך להגדיל עוד יותר את אורכו של נתיב הקוד. בנוסף לכך, יצירת קוד נפגעת בסופו של דבר כאשר אתה מציב את כל משפטי try הנחוצים, כי הממטב (optimizer) מוגבל במידת מה בנוכחות קוד לטיפול בחריגים – זה נכון גם לגבי קוד C++ שאינו מנוהל. גם אין כל ערך בכך שכל אובייקט גדול בארבעה בתים בגלל שדה נוסף מסוג ref עבור הספירה, מה שמגדיל את צריכת הזיכרון ואת גודל קבוצת העבודה (working set).

כדוגמה נציג שני יישומים המדגימים את המחיר הכרוך בכך. יישומים אלה ניתנים באדיבות צוות NET. בוחן ביצועים מסוים זה מבצע לולאה תוך הקצאת אובייקטים, ביצוע שתי הצבות, ויציאה מטווח ההכרה של ייחוס אחד. כמו עם כל בוחן ביצועים אחר, הוא נתון לפירוש סובייקטיבי. מישהו יכול לטעון שביחס לשגרה זו, ספירת הייחוסים ברובה ניתנת למיטוב (optimization). הדבר אולי נכון, אך כאן הכוונה היא להדגים את ההשפעה. בתוכנית אמיתית, מיטובים מסוגים אלה קשים, ואף בלתי אפשריים לביצוע. למעשה, ב- C++ מתכנתים מבצעים מיטובים אלה באופן ידני, והדבר גורם לטעויות בספירת ייחוסים. לכן, שים לב שבתוכנית אמיתית היחס בין הצבה להקצאה יהיה גבוה בהרבה מזה שמוצג כאן.

הנה הדוגמה הראשונה ref_gc.cs. גירסה זו מסתמכת על מעקב Garbage Collector (tracing GC):

```
using System;

public class Foo

{
   private static Foo m_f;
   private int m_member;

   public static void Main(String[] args)
   {
      int ticks = Environment.TickCount;

      Foo f2 = null;

      for (int i=0; i < 10000000; ++i)
      {
         Foo f = new Foo();

         // Assign static to f2.
         f2 = m_f;

         // Assign f to the static.
         m_f = f;

         // f goes out of scope.
      }

      // Assign f2 to the static.
      m_f = f2;

      // f2 goes out of scope.

      ticks = Environment.TickCount - ticks;
      Console.WriteLine("Ticks = {0}", ticks);
   }

   public Foo()
   {
   }
}
```

הנה הדוגמה השנייה, ref_rm.cs, שהיא גירסה אחרת לספירת ייחוסים ומשתמשת בפעולות נעילה להבטחת מטלות (thread safety):

```
using System;
using System.Threading;

public class Foo
{
   private static Foo m_f;
   private int m_member;
   private int m_ref;

   public static void Main(String[] args)
   {
      int ticks = Environment.TickCount;

      Foo f2 = null;

      for (int i=0; i < 10000000; ++i)
      {
         Foo f = new Foo();

         // Assign static to f2.
         if (f2 != null)
         {
            if (Interlocked.Decrement(ref f2.m_ref) == 0)
               f2.Dispose();
         }
         if (m_f != null)
            Interlocked.Increment(ref m_f.m_ref);
         f2 = m_f;

         // Assign f to the static.
         if (m_f != null)
         {
            if (Interlocked.Decrement(ref m_f.m_ref) == 0)
               m_f.Dispose();
         }
         if (f != null)
            Interlocked.Increment(ref f.m_ref);
         m_f = f;

         // f goes out of scope.
         if (Interlocked.Decrement(ref f.m_ref) == 0)
            f.Dispose();
      }
```

```
        // Assign f2 to the static.
        if (m_f != null)
        {
            if (Interlocked.Decrement(ref m_f.m_ref) == 0)
                m_f.Dispose();
        }
        if (f2 != null)
            Interlocked.Increment(ref f2.m_ref);
        m_f = f2;

        // f2 goes out of scope.
        if (Interlocked.Decrement(ref f2.m_ref) == 0)
            f2.Dispose();

        ticks = Environment.TickCount - ticks;
        Console.WriteLine("Ticks = {0}", ticks);
    }

    public Foo()
    {
        m_ref = 1;
    }

    public virtual void Dispose()
    {
    }
}
```

שים לב שיש רק מטלה אחת, ולכן אין התמודדות על אפיק הנתונים (bus contention). כך, דוגמה זו מציגה מקרה "אידיאלי". תוכל אולי לכוונן אותה, אך לא בהרבה. אין גם ערך לכך ש-Visual Basic לא היה צריך לטפל בפעולות נעילה (interlock) עבור ספירת הייחוסים שלו (בניגוד ל- ++C). בגרסאות קודמות של Visual Basic, רכיב פעל במבנה של מטלה בודדת והיה מובטח שרק מטלה אחת בכל זמן תתבצע בו. אחד מהיעדים של Visual Basic.NET הוא תכנות מרובה-מטלות (multithread programming). יעד אחר הוא להיפטר מהמורכבות של מודלים מרובי-מטלות של COM. עם זאת, לאלה מביניכם הרוצים לראות גירסה שאינה משתמשת בקידומות נעילה מוצגת הדוגמה הבאה ref_rs.cs, גם היא באדיבות צוות Net. דוגמה זו היא גירסה של ספירת ייחוסים, המניחה שהיא פועלת בסביבה של מטלה בודדת. תוכנית זו אינה איטית כמו הגירסה מרובת-המטלות, אך עדיין היא יותר איטית מגרסת GC.

```
using System;

public class Foo
```

```
{
    private static Foo m_f;
    private int m_member;
    private int m_ref;

    public static void Main(String[] args)
    {
        int ticks = Environment.TickCount;

        Foo f2 = null;

        for (int i=0; i < 10000000; ++i)
        {
            Foo f = new Foo();


            // Assign static to f2.
            if (f2 != null)
            {
                if (--f2.m_ref == 0)
                    f2.Dispose();
            }
            if (m_f != null)
                ++m_f.m_ref;
            f2 = m_f;

            // Assign f to the static.
            if (m_f != null)
            {
                if (--m_f.m_ref == 0)
                    m_f.Dispose();
            }
            if (f != null)
                ++f.m_ref;
            m_f = f;

            // f goes out of scope.
            if (--f.m_ref == 0)
                f.Dispose();
        }
```

```
        // Assign f2 to the static.
        if (m_f != null)
        {
            if (--m_f.m_ref == 0)
                m_f.Dispose();
        }
        if (f2 != null)
            ++f2.m_ref;
        m_f = f2;

        // f2 goes out of scope.
        if (--f2.m_ref == 0)
            f2.Dispose();

        ticks = Environment.TickCount - ticks;
        Console.WriteLine("Ticks = {0}", ticks);
    }

    public Foo()
    {
        m_ref = 1;
    }

    public virtual void Dispose()
    {
    }
}
```

כמובן שיש כאן מספר רב של משתנים. למרות זאת, תוצאות ההרצה של כל שלושת היישומים מראות שגרסת GC מתבצעת כמעט כפליים מהר יותר מהדוגמה של ספירת ייחוסים למטלה בודדת, ופי ארבע מהר יותר מהדוגמה של ספירת ייחוסים עם מקדמי נעילה (לסביבה מרובת-מטלות). להלן התוצאות שקיבלתי – שים לב שמספרים אלה מייצגים את הממוצע הפשוט של הרצות על מחשב IBM ThinkPad 570 המפעיל את המהדר NET Framework SDK Beta 1:

| | |
|---|---|
| GC Version (ref_gc) | 1162ms |
| Ref Counting (ref_rm)(multi-threaded) | 4757ms |
| Ref Counting (ref_rs)(single-threaded) | 1913ms |

## הפתרון המושלם

רבים יסכימו שהפתרון המושלם לבעיה זו היא מערכת שבה כל אובייקט הוא זול להקצאה, לשימוש ולאוסף. במערכת אידיאלית כזו, כל אובייקט מסיים באופן מסודר דטרמיניסטי, ברגע שהמתכנת מאמין שאובייקט אינו נחוץ יותר (מבלי להתחשב באם קיימים ייחוסים מעגליים). לאחר שהשקיע שעות רבות ללא ספור, צוות NET מאמין

שהדרך היחידה להשיג זאת היא על ידי שילוב של מעקב GC וספירת ייחוסים. נתוני בוחן הביצועים מראים שספירת ייחוסים היא יקרה מדי בשימוש רגיל עבור כל האובייקטים בסביבת התכנות. מסלולי הקוד ארוכים יותר, ומערך העבודה של קוד ונתונים גדול יותר. אם אתה משלב מחיר גבוה זה עם תוספת העלות של אוסף מעקב בכדי לשחרר ייחוסים מעגליים, אתה משלם מחיר מופרז עבור ניהול זיכרון.

יש לציין שבתחילת הפיתוח של NET נחקרו טכניקות שונות בכדי למצוא דרך לשפר את הביצועים של ספירת ייחוסים. לפני כן היו מספר מערכות עם ביצועים גבוהים במידה סבירה, הבנויים על ספירת ייחוסים. למרבה הצער, לאחר בחינת החומר הכתוב נקבע שמערכות אלו השיגו ביצועים משופרים על ידי ויתור על חלק מהדטרמיניסטיות שלהן.

כהערת אגב, יתכן ואין כל ערך לכך שתוכניות C++/COM אינן סובלות מבעיה זו. רוב תוכניות C++ עם ביצועים גבוהים המשתמשות ב-COM, מנצלות מחלקות C++ באופן פנימי, והמתכנת צריך לנהל את הזיכרון בצורה מפורשת. מתכנתי C++ בדרך כלל משתמשים ב-COM רק כדי ליצור את הממשקים עבור לקוחותיהם. זו תכונת מפתח, המאפשרת לתוכניות אלו לפעול כראוי. עם זאת, אין זה כמובן היעד של NET, שבה הסוגיות של ניהול זיכרון מיועדות להיות מטופלות באמצעות Garbage Collector, ולא על ידי המתכנת.

## פתרון שהוא (כמעט) מושלם

כמו בחיים, אינך יכול לקבל את הכל ביחס לפתרון ניהול המשאבים. אך מה אם היית יכול לקבל גמר דטרמיניסטי רק עבור האובייקטים הרצויים? צוות NET השקיע מחשבה מרובה בנושא זה. זכור זאת בהקשר להעתקה המדויקת של סמנטיקת Visual Basic 6 אל המערכת החדשה. רוב הניתוחים עדיין ישימים, אך מספר רעיונות שהוזנחו בעבר נראים כעת נוחים יותר כטכניקות לניהול משאבים, במקום סמנטיקת אורך חיים שקופה של Visual Basic 6.

הניסיון הראשון היה לסמן מחלקה שזקוקה לגמר דטרמיניסטי באמצעות תכונה או על ידי הורשה ממחלקה "מיוחדת", בהנחה שהדבר יגרום לספירת ייחוסים עבור האובייקט. בעניין זה נחקרו מספר תוכניות שכללו הרחבות של System.Object יחד עם שינוי מחלקת הבסיס של היררכיית המחלקות, כדי ליצור מחלקה אחרת כלשהי אשר תכרוך את עולם ספירת הייחוסים יחד עם העולם האחר. לרוע המזל, היו מספר סוגיות שלא ניתן היה לחמוק מהם, כמתואר בסעיפים הבאים.

### Composition

כשלוקחים אובייקט שזקוק לגמר דטרמיניסטי ומאחסנים אותו באובייקט שאינו כזה, מאבדים את הדטרמיניסטיות (כי הדטרמיניזם אינו משתמע). הבעיה היא, שהדבר פוגע בדיוק בלב היררכיית המחלקות. לדוגמה, מה בדבר מערכים? אם אתה רוצה מערך של אובייקטים דטרמיניסטיים, גם המערכים צריכים להיות כאלה. ומה בדבר אוספים וטבלאות hash? הרשימה הולכת ותופחת – עוד לפני שתהיה מודע לכך, ספריית המחלקות תחייב ספירת ייחוסים, וכך תחמיץ את המטרה.

אפשרות אחרת היא לפצל את ספריית המחלקות של סביבת NET לשתי גרסאות של סוגי מחלקה רבים, כמו לדוגמה, מערכים דטרמיניסטים ומערכים שאינם דטרמיניסטיים. עניין זה נשקל בכובד ראש. עם זאת, ההחלטה הסופית היתה שעלול להיגרם בלבול בניהול של שני עותקים מלאים של סביבת NET. הביצוע בדרך זו יהיה גרוע כתוצאה מטעינה של שני עותקים מכל מחלקה, ובסופו של דבר – לא יהיה מעשי.

נבחנו פתרונות מסוימים למחלקות מסוימות, אך אף אחד מפתרונות אלה לא יכול היה לפתור את השימוש הכללי הניתן להרחבה עבור סביבת NET כולה. כמו כן, שקלו להפוך את הבעיה הבסיסית – אובייקט שאינו דטרמיניסטי המכיל ייחוס לאובייקט דטרמיניסטי – לשגיאה, אך הגיעו למסקנה שהגבלה זו תכביד על כתיבת תוכניות ממשיות.

### Casting

קיימת סוגיה הקשורה להמרה (casting), ובה יש להתייחס לשאלות הבאות: האם ניתן להמיר אובייקט דטרמיניסטי ל-System.Object? אם כן, האם תהיה ספירת ייחוס עבור האובייקט? אם התשובה היא כן, כל דבר ייכלל בספירת הייחוסים. אם התשובה היא לא, האובייקט מאבד את הדטרמיניסטיות שלו. אם התשובה היא "שגיאה", יש כאן הפרה של הנחת היסוד ש-System.Object הוא המקור של היררכיית האובייקטים.

### Interfaces

עד כאן היית צריך להבין עד כמה מורכבת סוגיה זו. אם אובייקט דטרמיניסטי מיישם ממשקים, האם הייחוס המסווג כייחוס ממשק נספר גם כן? אם התשובה היא כן, אתה סופר ייחוסים עבור כל האובייקטים המיישמים ממשקים (שים לב שהסוג System.Int32 מיישם ממשקים). אם התשובה היא לא, האובייקט מאבד את הדטרמיניסטיות שלו. אם התשובה היא "שגיאה", אזי אובייקטים דטרמיניסטיים אינם יכולים ליישם ממשקים. אם התשובה היא "תלוי אם הממשק מסומן כדטרמיניסטי", יש כאן בעיה נוספת של הסתעפות. ממשקים אינם אמורים להכתיב את הסמנטיקה של אורך חיי האובייקט. עם זאת, מה קורה כאשר מישהו מיישם API עם ממשק Icollection, והאובייקט שלך המיישם אותו זקוק לדטרמיניסטיות, אך הממשק לא הוגדר כך? במקרה זה לא יהיה לך מזל. בתסריט זה צריך להגדיר שני ממשקים, אחד דטרמיניסטי והשני לא-דטרמיניסטי, וליישם כל שיטה פעמיים. תאמין או לא, האפשרות לייצר באופן אוטומטי שתי גרסאות של השיטות נשקלה אף היא. קו מחשבה זה לא הניב תוצאה והרעיון ננטש.

## Dispose Design Pattern

היכן כל זה משאיר אותנו? נראה שנותרנו כרגע עם גמר דטרמיניסטי הפועל באופן הבא:

- Garbage Collector עוקב אחר אובייקטים שיש להם ייחוסים.
- Garbage Collector מפעיל תמיד מטלה בעלת קדימות נמוכה הסורקת את האובייקטים שלו, כדי לקבוע מתי לאובייקט אין יותר ייחוס.
- מטלה שנייה בעלת קדימות נמוכה אחראית לאחר מכן לניקוי. מטלה זו קוראת לשיטה finalize של האובייקט.

גישה זו אכן פותרת את בעיית הייחוס המעגלי, אך גורמת לבעיות אחרות. לדוגמה, אין הבטחה לכך שהשיטה finalize תיקרא על ידי זמן הריצה! זו בדיוק המשמעות של המילה "דטרמיניסטי" ב"גמר דטרמיניסטי". נשאלת השאלה, איך אוכל לבצע את הניקוי הדרוש ולדעת שקוד הניקוי הנכון נקרא בכל פעם?

במקרה של שחרור (או פינוי) של משאבים יקרים, מיקרוסופט מציעה פתרון המבוסס על תבנית התכנון Dispose. תבנית תכנון זו ממליצה שהאובייקט יחשוף שיטה ציבורית בעלת שם כללי כמו Cleanup או Dispose. על המשתמש לקרוא לה כאשר הוא מסיים להשתמש באובייקט, ומתכנן המחלקה צריך לכלול בשיטה זו כל פעולת ניקוי נדרשת. למעשה תיווכח שמחלקות רבות בספריית המחלקות של סביבת NET מיישמות את השיטה Dispose למטרה זו. לדוגמה, התיעוד למחלקה System.Winforms.TrayIcon אומר: "קרא ל-Dispose כאשר אתה מסיים להשתמש ב-TrayIcon".

# Inheritance

כפי שציינתי בפרק 1, משתמשים ב**הורשה** (inheritance) כאשר מחלקה נבנית על מחלקה אחרת (במונחים של נתונים או התנהגות) ומצייתת לחוקים של יכולת המרה. כלומר, המחלקה הנגזרת יכולה להוות תחליף למחלקת הבסיס. דוגמה לכך יכולה להיות היררכיה של מחלקות מסד נתונים. נניח שאתה רוצה מחלקה שתטפל במסדי נתונים של Microsoft SQL Server ושל Oracle. מכיון שמסדי נתונים אלה נבדלים זה מזה במספר היבטים, תרצה לתכנן מחלקה שונה עבור כל אחד מהם. עם זאת, למסדי נתונים אלה יש פונקציונליות משותפת רבה, ולכן תרצה להכניס פונקציונליות זו לתוך מחלקת בסיס, לגזור את שתי המחלקות האחרות ממחלקה זו, ולפעמים לדרוס או לשנות את התנהגות מחלקת הבסיס שעברה בהורשה.

בכדי לרשת מחלקה אחת מאחרת, עליך להשתמש בתחביר הבא:

```
class <derivedClass>: <baseClass>
```

כך תיראה דוגמת מסדי נתונים זו:

```
using System;

class Database
{
   public Database()
   {
      CommonField = 42;
   }
```

```
    public int CommonField;

    public void CommonMethod()
    {
        Console.WriteLine("Database.Common Method");
    }
}

class SQLServer : Database
{
    public void SomeMethodSpecificToSQLServer()
    {
        Console.WriteLine("SQLServer.SomeMethodSpecificToSQLServer");
    }
}

class Oracle : Database
{
    public void SomeMethodSpecificToOracle()
    {
        Console.WriteLine("Oracle.SomeMethodSpecificToOracle");
    }
}

class InheritanceApp
{
    public static void Main()
    {
        SQLServer sqlserver = new SQLServer();

        sqlserver.SomeMethodSpecificToSQLServer();
        sqlserver.CommonMethod();
        Console.WriteLine("Inherited common field = {0}",
                          sqlserver.CommonField);
    }
}
```

הידור וביצוע של התוכנית מפיק את הפלט הבא:

```
SQLServer.SomeMethodSpecificToSQLServer
Database.Common Method
Inherited common field = 42
```

שים לב שהשיטות Database.CommonMethod ו-Database.CommonField הן כעת חלק מהגדרת המחלקה SQLServer. מכיון שהמחלקות SQLServer ו-Oracle נגזרות ממחלקת הבסיס Database, הן יורשות ממנה כמעט את כל החברים המוגדרים public,

protected ו-internal. יוצא מהכלל הוא הבנאי, אשר אינו יכול לעבור בהורשה. כל מחלקה חייבת ליישם בנאי משלה ללא תלות או קשר למחלקת הבסיס שלה.

דריסת שיטות נסקרת בפרק 6. עם זאת רק אציין, שדריסת שיטות מאפשרת לרשת (inherit) שיטה ממחלקת בסיס ולאחר מכן לשנות את המימוש שלה. מחלקות מופשטות (abstract classes) כבולות במידה רבה לדריסת שיטות, וגם בהן נדון בפרק 6.

## Multiple Interfaces

ברצוני להבהיר סוגיה זו, כי הורשה מרובה הפכה להיות עניין שנוי במחלוקת בקבוצות דיון וברשימות דיוור רבות. #C אינה תומכת בהורשה מרובה באמצעות גזירה (derivation). עם זאת, תוכל לקבץ את תכונות ההתנהגות של יישויות תכנות מרובות על ידי יישום ממשקים מרובים (multiple interfaces). פרק 9 עוסק בממשקים ותמצא בו הסבר איך להשתמש בהם. כעת חשוב על ממשקי #C כעל ממשק COM.

מכאן, התוכנית הבאה אינה חוקית:

```
class Foo
{
}

class Bar
{
}

class MITest : Foo, Bar
{
   public static void Main ()
   {
   }
}
```

השגיאה שתקבל בדוגמה זו קשורה לאופן שבו אתה מיישם ממשקים. את הממשקים שאתה בוחר ליישם עליך לפרט לאחר מחלקת הבסיס. לפיכך, בדוגמה זו מהדר #C חושב ש-Bar צריך להיות מסוג ממשק. זו הסיבה שמהדר #C יציג את הודעת השגיאה הזו:

```
'Bar' : type in interface list is not an interface
```

הדוגמה הבאה (מציאותית יותר) חוקית לגמרי, כי המחלקה MyFancyGrid נגזרת מהמחלקה Control וממשת את הממשקים ISerializable ו-IDataBound:

```
class Control
{
}

interface ISerializable
{
}
```

```
interface IDataBound
{
}

class MyFancyGrid : Control, ISerializable, IDataBound
{
}
```

הנקודה כאן היא, שהדרך היחידה בה אתה יכול לממש משהו הדומה להורשה מרובה ב-#C היא באמצעות ממשקים (interfaces).

## Sealed Classes

אם ברצונך להבטיח שמחלקה לא תשמש לעולם כמחלקת בסיס, השתמש במציין **sealed** (חתום) בעת הגדרת המחלקה. ההגבלה היחידה היא שמחלקה מופשטת (abstract class) אינה יכולה לשמש כמחלקה חתומה, כי מחלקות מופשטות נועדו לשמש מטבען כמחלקות בסיס. כמו כן, למרות שמטרת מחלקה חתומה היא למנוע גזירה לא מכוונת, מיטובים מסוימים של זמן ריצה מתאפשרים כתוצאה מהגדרת מחלקה כחתומה. ליתר דיוק, מכיון שהמהדר מבטיח שלמחלקה לא תהיינה מחלקות נגזרות לעולם, אפשר להפוך קריאות לפונקציה וירטואלית החברה במופעים של מחלקה חתומה לקריאות לא וירטואליות. הנה דוגמה של מחלקה חתומה:

```
using System;
sealed class MyPoint
{
   public MyPoint(int x,int y)
   {
      this.x =x;
      this.y =y;
   }

   private int X;
   public int x
   {
      get
      {
         return this.X;
      }
      set
      {
         this.X =value;
      }
   }
```

```
    private int Y;
    public int y
    {
        get
        {
            return this.Y;
        }
        set
        {
            this.Y = value;
        }
    }
}

class SealedApp
{
    public static void Main()
    {
        MyPoint pt = new MyPoint(6,16);
        Console.WriteLine("x = {0}, y = {1}", pt.x, pt.y);
    }
}
```

שים לב שהשתמשתי במציין הגישה private בהצהרה של חברי המחלקה הפנימיים X ו-Y. שימוש במציין protected גורם למהדר להדפיס אזהרה בגלל העובדה שחברים מוגנים זמינים למחלקות נגזרות. כפי שאתה כבר יודע, למחלקות חתומות אין מחלקות נגזרות כלשהן.

# סיכום

המושג של מחלקות והקשר שלהן לאובייקטים הוא הבסיס לתפישת התכנות המושתת על אובייקטים. התכונות מוכוונות-העצמים של #C בנויות על תכונות ההורשה של ++C ומעוצבים ומשופרים על ידי התכונות של סביבת NET. במערכות מנוהלות, כמו סביבת הריצה CLR, ניהול משאבים הוא סוגיה מדאיגה למדי עבור מפתחים. CLR שואף לשחרר את המתכנתים מהעבודה האפורה של ספירת ייחוסים באמצעות garbage collection המבוסס על גמר דטרמיניסטי (deterministic finalization). בנוסף לכך, הורשה ב-#C מטופלת באופן שונה מאשר ב- ++C. למרות שיש תמיכה רק בהורשה יחידה, המפתחים יכולים לממש כמה מהיתרונות של הורשה מרובה (multiple inheritance) באמצעות מימוש של ממשקים מרובים (multiple interfacess).

# פרק 6 שיטות

כפי שלמדת בפרק 1 העוסק ביסודות התכנות מוכוון-עצמים, **מחלקות** הן חבילות מכומסות (encapsulated) של **נתונים** ו**שיטות** הפועלות על נתונים אלה. במילים אחרות, **שיטות** (methods) מספקות למחלקות את תכונות ההתנהגות, ואנו מכנים אותן על פי הפעולות שאנו רוצים שתבצענה למעננו. בפרק זה נדון בסוגיות מפורטות הקשורות להגדרה ולקריאה של שיטות ב-#C. כאן תגלה את מילות המפתח ref ו-out של פרמטרים לשיטות ואיך הן מאפשרות להגדיר שיטה שתחזיר יותר מערך אחד אל השיטה הקוראת לה. גם תלמד איך להגדיר שיטות מועמסות, כך ששיטות מרובות עם שם זהה יכולות לשנות את תפקודן כתלות בסוגים ו/או מספר הארגומנטים המועברים אליהן. לאחר מכן תלמד איך להתמודד עם מצבים שבהם אינך יודע מהו המספר המדויק של ארגומנטים שיהיו לשיטה עד לזמן הריצה. לסיום הפרק, נדון בשיטות וירטואליות על בסיס הדיון בהורשה בפרק 5, ואיך להגדיר שיטות סטטיות.

## הפרמטרים ref ו-out

כאשר מנסים לקבל מידע באמצעות שיטה ב-#C, ניתן לקבל ערך מוחזר אחד בלבד. לכן מתקבל הרושם שניתן לקבל ערך אחד בלבד לכל קריאה לשיטה. כמובן שקריאה לשיטה אחת עבור כל פיסת מידע נדרשת מכבידה במקרים רבים. לדוגמה, נניח שקיימת מחלקה Color המייצגת צבע נתון בעל שלושה ערכים באמצעות המודל הסטנדרטי RGB (red-green-blue) לתיאור צבעים. השימוש בערך מוחזר בודד מחייב אותנו לכתוב קוד בדומה לזה, כדי לקבל את כל שלושת הצבעים:

```
// Assuming color is an instance of a Color class.
int red = color.GetRed();
int green = color.GetGreen();
int blue = color.GetBlue();
```

למעשה, אנו רוצים משהו דומה לזה:

```
int red;
int green;
int blue;
color.GetRGB(red, green, blue);
```

עם זאת, יש כאן בעיה. כאשר קוראים לשיטה color.GetRGB, הערכים עבור הארגומנטים red, green ו-blue מועתקים למחסנית המקומית של השיטה, וכל שינוי שיבוצע על ידי השיטה לא יחול על משתני הקוד הקורא.

ב- C++ עוקפים בעיה זו על ידי העברת מצביעים או ייחוסים למשתנים, כך שהשיטה פועלת על נתוני השיטה הקוראת. הפתרון ב-C# דומה. למעשה, C# מציעה שני פתרונות דומים. הראשון משתמש במילת המפתח **ref**, אשר אומרת למהדר C# שהארגומנטים המועברים מצביעים אל המקום בזיכרון שבו נמצאים המשתנים בקוד של השיטה הקוראת. בדרך זו, אם השיטה הנקראת משנה ערכים אלה וחוזרת, משתני השיטה הקוראת ישתנו אף הם. הקוד הבא מראה איך להשתמש ב-ref בדוגמה של המחלקה Color:

```
using System;

class Color
{
   public Color()
   {
      this.red = 255;
      this.green = 0;
      this.blue = 125;
   }

   protected int red;
   protected int green;
   protected int blue;

   public void GetColors(ref int red, ref int green, ref int blue)
   {
      red = this.red;
      green = this.green;
      blue = this.blue;
   }
}
```

```
class RefTest1App
{
   public static void Main()
   {
      Color color = new Color();
      int red;
      int green;
      int blue;
      color.GetColors(ref red, ref green, ref blue);
      Console.WriteLine("red = {0}, green = {1}, blue = {2}",
            red, green, blue);
   }
}
```

לשימוש במילת המפתח ref יש מגרעת אחת שבגללה הידור הקוד הקודם ייכשל. כאשר משתמשים במילת המפתח ref חייבים לאתחל את הארגומנטים המועברים לפני הקריאה לשיטה. לפיכך, כדי שהקוד הקודם יפעל יש לשנותו באופן הבא:

```
using System;

class Color
{
   public Color()
   {
      this.red = 255;
      this.green = 0;
      this.blue = 125;
   }

   protected int red;
   protected int green;
   protected int blue;

   public void GetColors(ref int red, ref int green, ref int blue)
   {
      red = this.red;
      green = this.green;
      blue = this.blue;
   }
}
```

```
class RefTest2App
{
   public static void Main()
   {
      Color color = new Color();
      int red = 0;
      int green = 0;
      int blue = 0;
      color.GetColors(ref red, ref green, ref blue);
      Console.WriteLine("red = {0}, green = {1}, blue = {2}",
            red, green, blue);
   }
}
```

אתחול המשתנים כאן נראה מיותר, כי ממילא נכתוב עליהם. לכן #C מספקת פתרון נוסף שהוא אמצעי חליפי להעברת ארגומנט אשר ערכו לאחר שינוי צריך להיראות לקוד הקורא. אמצעי זה הוא מילת המפתח **out**. הנה שוב דוגמה של המחלקה Color עם שימוש במילת המפתח out:

```
using System;

class Color
{
   public Color()
   {
      this.red = 255;
      this.green = 0;
      this.blue = 125;
   }

   protected int red;
   protected int green;
   protected int blue;

   public void GetColors(out int red, out int green, out int blue)
   {
      red = this.red;
      green = this.green;
      blue = this.blue;
   }
}
```

```
class OutTest1App
{
   public static void Main()
   {
      Color color = new Color();
      int red;
      int green;
      int blue;
      color.GetColors(out red, out green, out blue);
      Console.WriteLine("red = {0}, green = {1}, blue = {2}",
               red, green, blue);
   }
}
```

ההבדל היחיד בין מילות המפתח ref ו-out הוא בכך ש-out אינה מחייבת את הקוד הקורא לשיטה לאתחל תחילה את הארגומנטים המועברים. מתי, אם כן, שימושית מילת המפתח ref? משתמשים בה כאשר רוצים להיות בטוחים שהשיטה הקוראת אתחלה את הארגומנטים. בדוגמה הקודמת ניתן להשתמש ב-out, כי השיטה הנקראת לא היתה תלויה בערך המשתנה המועבר. אך מה קורה כאשר השיטה הנקראת משתמשת בערך של פרמטר? התבונן בקוד הבא:

```
using System;

class Window
{
   public Window(int x, int y)
   {
      this.x = x;
      this.y = y;
   }

   protected int x;
   protected int y;

   public void Move(int x, int y)
   {
      this.x = x;
      this.y = y;
   }

   public void ChangePos(ref int x, ref int y)
   {
      this.x += x;;
      this.y += y;
```

```
        x = this.x;
        y = this.y;
    }
}

class OutTest2App

{
    public static void Main()
    {
        Window wnd = new Window(5, 5);

        int x = 5;
        int y = 5;

        wnd.ChangePos(ref x, ref y);
        Console.WriteLine("{0}, {1}", x, y);

        x = -1;
        y = -1;
        wnd.ChangePos(ref x, ref y);
        Console.WriteLine("{0}, {1}", x, y);
    }
}
```

כפי שתוכל לראות, השיטה Window.ChangePos פועלת על סמך הערכים המועברים אליה. במקרה זה, מילת המפתח ref מאלצת את השיטה הקוראת לאתחל את הערך, כדי שתוכל לפעול כראוי.

# Method Overloading

**העמסת שיטות** (method overloading) מאפשרת למתכנת #C להשתמש בשם זהה עבור שיטות מרובות שיש ביניהן הבדל בארגומנטים המועברים בלבד. יכולת זו שימושית ביותר, לפחות בשני תסריטים. התסריט הראשון מתייחס למצבים שבהם רוצים לחשוף שיטה אחת ויחידה כאשר ההתנהגות שלה משתנה במעט בהתאם לסוגי הערכים המועברים. לדוגמה, נניח שיש מחלקה המאפשרת ליישום שלך לכתוב מידע אבחון בדיסק. בכדי להגמיש במעט את השיטה, אולי תרצה להפעיל כמה צורות של השיטה Write עבור הנתונים שיש לרשום. מלבד קבלת מחרוזת הנתונים שיש לכתוב בדיסק, השיטה גם תוכל לקבל זיהוי של המשאב שממנו התקבלה מחרוזת הנתונים. ללא היכולת להעמיס שיטות, עליך ליישם שיטה לכל אחד ממצבים אלה, כמו למשל WriteString ו-WriteFromResourceID.

עם זאת, העמסת שיטות מאפשרת לממש את השיטות הבאות, הנקראות שתיהן WriteEntry, אך הן נבדלות זו מזו בסוג הפרמטר בלבד:

```
using System;

class Log
{
   public Log(string fileName)
   {
      // Open fileName and seek to end.
   }

   public void WriteEntry(string entry)
   {
      Console.WriteLine(entry);
   }

   public void WriteEntry(int resourceId)
   {
      Console.WriteLine
     ("Retrieve string using resource id and write to log");
   }
}

class Overloading1App
{
   public static void Main()
   {
      Log log = new Log("My File");
      log.WriteEntry("Entry one");
      log.WriteEntry(42);
   }
}
```

תסריט אחר, שבו העמסת שיטות שימושית מאוד הוא כאשר משתמשים בבנאים, שהינם שיטות הנקראות כאשר יוצרים מופע של אובייקט. נניח שרוצים ליצור מחלקה הניתנת לבנייה ביותר מדרך אחת, כמו לדוגמה על ידי העברת ידית לקובץ (ערך int) או על ידי העברת שם קובץ (ערך string) בכדי לפתוח קובץ. מכיון שחוקי C# מכתיבים ששם בנאי המחלקה יהיה כשם המחלקה עצמה, לא ניתן ליצור בפשטות שיטות בעלות שמות שונים לכל אחד מסוגי המשתנים. במקום זאת, יש להעמיס את הבנאי:

```
using System;

class File
{
}
```

```
class CommaDelimitedFile
{
   public CommaDelimitedFile(String fileName)
   {
      Console.WriteLine("Constructed with a file name");
   }

   public CommaDelimitedFile(File file)
   {
      Console.WriteLine("Constructed with a file object");
   }
}

class Overloading2App
{
   public static void Main()
   {
      File file = new File();
      CommaDelimitedFile file2 = new CommaDelimitedFile(file);
      CommaDelimitedFile file3 =
new CommaDelimitedFile("Some file name");
   }
}
```

נקודה חשובה שיש לזכור בנוגע להעמסת שיטות היא שרשימת הארגומנטים של כל שיטה חייבת להיות שונה. לפיכך, הקוד הבא לא יעבור הידור, כי השוני היחיד בין שתי גרסאות השיטה Overloading3App.Foo הוא הסוג המוחזר:

```
using System;

class Overloading3App
{
   void Foo(double input)
   {
      Console.WriteLine("Overloading3App.Foo(double)");
   }

   // ERROR: Only differs by return value. Won't compile.
   double Foo(double input)
   {
      Console.WriteLine("Overloading3App.Foo(double)
                  (second version)");
   }
```

```
    public static void Main()
    {
        Overloading3App app = new Overloading3App();

        double i = 5;
        app.Foo(i);
    }
}
```

# שיטה עם מספר לא ידוע של פרמטרים

לעיתים, מספר הארגומנטים שיועברו לשיטה אינו ידוע עד לזמן הריצה. לדוגמה, נניח שרוצים מחלקה המשרטטת גרף בהתאם לסדרה של קואורדינטות x ו-y. המחלקה היתה יכולה לחשוף שיטה עם ארגומנט בודד מסוג אובייקט Point (נקודה) יחיד המייצג את הערכים x ו-y גם יחד. שיטה זו תאחסן לאחר מכן כל אובייקט Point ברשימה מקושרת או בחבר מסוג מערך עד לרגע שבו השיטה הקוראת תרצה להדפיס את כל רצף הנקודות. עם זאת, התכנון בגישה זו לקוי מכמה סיבות. ראשית, המשתמש נדרש לבצע את העבודה המיותרת של קריאה לשיטה אחת לסימון הנקודות של הקו שיש לשרטט – מלאכה מייגעת למדי אם הקו יהיה ארוך – ואז לקרוא לשיטה אחרת בכדי לשרטט את הקו. שנית, המחלקה צריכה לאחסן את הנקודות בדרך מסוימת למרות שהן דרושות לשימוש של השיטה DrawLine בלבד. **פרמטרים משתנים** (variable parameters) פותרים בעיה מסוג זה באופן מושלם.

תוכל להגדיר מספר משתנה של פרמטרים לשיטה באמצעות מילת המפתח **params** והגדרת מערך ברשימת הארגומנטים. הנה דוגמה של המחלקה Draw המאפשרת למשתמש לבצע קריאה אחת שלוקחת מספר משתנה של אובייקטים מסוג Point ומדפיסה כל אחד מהם:

```
using System;

class Point
{
    public Point(int x, int y)
    {
        this.x = x;
        this.y = y;
    }

    public int x;
    public int y;
}
```

```
class Chart
{
   public void DrawLine(params Point[] p)
   {
      Console.WriteLine("\nThis method would print a line " +
                        "along the following points:");
      for (int i = 0; i < p.GetLength(0); i++)

      {
         Console.WriteLine("{0}, {1}", p[i].x, p[i].y);
      }
   }
}

class VarArgsApp
{
   public static void Main()
   {
      Point p1 = new Point(5,10);
      Point p2 = new Point(5, 15);
      Point p3 = new Point(5, 20);

      Chart chart = new Chart();
      chart.DrawLine(p1, p2, p3);
   }
}
```

השיטה DrawLine אומרת למהדר #C שהיא מסוגלת לקלוט מספר משתנה של אובייקטים מסוג Point. בזמן ריצה, השיטה משתמשת בלולאת for פשוטה בכדי לבצע איטרציה על אובייקטים מסוג Point המועברים אליה ולהדפיס כל אחד מהם.

שים לב שביישום אמיתי עדיף לגשת לחברים x ו-y של האובייקט Point דרך מאפיינים, במקום להגדירם כ-public. בנוסף לכך, יהיה עדיף להשתמש בלולאת foreach בשיטה DrawLine במקום בלולאת for. עם זאת, בכדי לא לפגוע ברצף הלימוד של הספר, לא הצגתי עדיין תכונות אלו של השפה. נדון במאפיינים בפרק 7, ובמשפט foreach נדון בפרק 11.

# Virtual Methods

בפרק 5 ראינו שניתן לגזור מחלקה אחת ממחלקה אחרת, כך שמחלקה אחת תוכל לרשת מחלקה אחרת ולהיבנות על היכולות שלה. מכיון שטרם עסקנו בשיטות, הדיון נגע בהורשה של שדות ושיטות בלבד. במילים אחרות, לא בחנתי את היכולת לשנות את התנהגות מחלקת הבסיס במחלקה הנגזרת. יכולת זו ממומשת באמצעות **שיטות וירטואליות** (virtual methods).

## Method Overriding

הבה נראה תחילה איך דורסים (override) את הפונקציונליות של מחלקת הבסיס בשיטה שעברה בהורשה. נתחיל במחלקת בסיס המייצגת עובד. בכדי לשמור על פשטות הדוגמה, נכלול במחלקה זו שיטה יחידה הנקראת CalculatePay, שאינה עושה דבר מלבד למסור לנו את שם השיטה שנקראה. הדבר יעזור לקבוע מאוחר יותר איזו שיטה נקראה מעץ ההורשה.

```
class Employee
{
   public void CalculatePay()
   {
      Console.WriteLine("Employee.CalculatePay()");
   }
}
```

נניח שרוצים לגזור מחלקה מ-Employee ולדרוס את השיטה CalculatePay, כך שתבצע משהו ייחודי למחלקה הנגזרת. לשם כך, יש להשתמש במילת המפתח new בהגדרת השיטה של המחלקה הנגזרת. הנה הקוד הממחיש עד כמה פשוט הדבר:

```
using System;

class Employee
{
   public void CalculatePay()
   {
      Console.WriteLine("Employee.CalculatePay()");
   }
}

class SalariedEmployee : Employee
{
   // The new keyword enables you to override the
   // base class' implementation.
   new public void CalculatePay()
   {
      Console.WriteLine("SalariedEmployee.CalculatePay()");
   }
}
```

```
class Poly1App
{
   public static void Main()
   {
      Poly1App poly1 = new Poly1App();

      Employee baseE = new Employee();
      baseE.CalculatePay();


      SalariedEmployee s = new SalariedEmployee();
      s.CalculatePay();
   }
}
```

לאחר הידור וביצוע היישום נקבל את הפלט הבא:

```
c:\>Poly1App
Employee.CalculatePay()
Salaried.CalculatePay()
```

## Polimorphism

דריסת שיטות באמצעות מילת המפתח new פועלת כראוי אם יש ייחוס לאובייקט הנגזר. עם זאת, מה קורה כאשר יש ייחוס משתמע (המרה כלפי מעלה) למחלקת בסיס ורוצים שהמהדר יקרא לשיטה של המחלקה הנגזרת? **רב-צורתיות** (polimorphism) מאפשרת להגדיר שיטה יותר מפעם אחת לאורך היררכיית המחלקות. הדבר נעשה כך שסביבת זמן הריצה קוראת לגירסה המתאימה של אותה שיטה בהתאם לאובייקט המדויק שבו משתמשים.

בכדי להבין את המשמעות נתבונן שוב בדוגמת העובד. היישום Poly1App מתבצע באופן תקין, כי יש לנו שני אובייקטים: אובייקט Employee ואובייקט SalariedEmployee. ביישום מעשי יותר, קרוב לוודאי שנקרא את כל רשומות העובד ממסד נתונים ונאחסן אותם במערך. למרות שחלק מהעובדים יכולים להשתכר לפי שעות ואחרים בשכר חודשי, נאחסן את כולם במערך מסוג Employee של המחלקה הבסיסית. עם זאת, כאשר אנו מבצעים איטרציה על מערך זה, מקבלים וקוראים לשיטה CalculatePay של כל אובייקט, אנו רוצים שהמהדר יקרא למימוש המתאים של השיטה CalculatePay עבור כל אובייקט.

בדוגמה הבאה הוספתי מחלקה חדשה, ContractEmployee. המחלקה הראשית של היישום מכילה כעת מערך מסוג Employee ושתי שיטות נוספות: LoadEmployee הטוענת את אובייקטי העובד לתוך המערך, ו-DoPayroll המבצעת איטרציה על המערך כשבכל לולאה היא קוראת לשיטה CalculatePay של כל אובייקט.

```
using System;

class Employee
{
   public Employee(string name)
   {
      this.Name = name;
   }

   protected string Name;

   public string name

   {
      get
      {
         return this.Name;
      }
   }

   public void CalculatePay()
   {
      Console.WriteLine("Employee.CalculatePay called for {0}", name);
   }
}

class ContractEmployee : Employee
{
   public ContractEmployee(string name)
   : base(name)
   {
   }

   public new void CalculatePay()
   {
      Console.WriteLine("ContractEmployee.CalculatePay called for {0}", name);
   }
}

class SalariedEmployee : Employee
{
   public SalariedEmployee (string name)
   : base(name)
   {
   }
```

```
    public new void CalculatePay()
    {
        Console.WriteLine("SalariedEmployee.CalculatePay called for {0}", name);
    }
}

class Poly2App
{
    protected Employee[] employees;

    public void LoadEmployees()
    {
        // Simulating loading from a database.
        employees = new Employee[2];
        employees[0] = new ContractEmployee("Kate Dresen");
        employees[1] = new SalariedEmployee("Megan Sherman");
    }

    public void DoPayroll()
    {
        foreach(Employee emp in employees)
        {
            emp.CalculatePay();
        }
    }

    public static void Main()
    {
        Poly2App poly2 = new Poly2App();
        poly2.LoadEmployees();
        poly2.DoPayroll();
    }
}
```

עם זאת, כאשר נבצע את היישום נקבל את הפלט הבא:

```
c:\>Poly2App
Employee.CalculatePay called for Kate Dresen
Employee.CalculatePay called for Megan Sherman
```

כמובן שזו אינה התוצאה הרצויה. השיטה CalculatePay אשר מימשנו במחלקת הבסיס נקראת עבור כל אובייקט. מה שקרה כאן הוא דוגמה לתופעה הנקראת **כריכה מוקדמת** (early binding). כאשר הקוד עבר הידור, מהדר #C ראה את הקריאה ל- emp.CalculatePay וקבע את כתובת הזיכרון שיש לדלג אליה כאשר מתבצעת הקריאה. במקרה זה, תהיה זו כתובת הזיכרון של השיטה Employee.CalculatePay.

התבונן בקוד MSIL הבא שנוצר מהיישום Poly2App, ושים לב במיוחד לשורה IL_0014 ולעובדה שהיא מבצעת קריאה מפורשת לשיטה Employee.CalculatePay:

```
.method public hidebysig instance void DoPayroll() il managed
{
  // Code size       34 (0x22)
  .maxstack  2
  .locals (class Employee V_0,
           class Employee[] V_1,
           int32 V_2,
           int32 V_3)

  IL_0000:  ldarg.0
  IL_0001:  ldfld      class Employee[] Poly2App::employees
  IL_0006:  stloc.1
  IL_0007:  ldloc.1
  IL_0008:  ldlen
  IL_0009:  conv.i4
  IL_000a:  stloc.2
  IL_000b:  ldc.i4.0
  IL_000c:  stloc.3
  IL_000d:  br.s       IL_001d
  IL_000f:  ldloc.1
  IL_0010:  ldloc.3
  IL_0011:  ldelem.ref
  IL_0012:  stloc.0
  IL_0013:  ldloc.0
  IL_0014:  call       instance void Employee::CalculatePay()
  IL_0019:  ldloc.3
  IL_001a:  ldc.i4.1
  IL_001b:  add
  IL_001c:  stloc.3
  IL_001d:  ldloc.3
  IL_001e:  ldloc.2
  IL_001f:  blt.s      IL_000f
  IL_0021:  ret
} // end of method Poly2App::DoPayroll
```

קריאה זו לשיטה Employee.CalculatePay היא הבעיה. במקום זאת, אנו רוצים שתהיה **כריכה מאוחרת** (late binding), שבה המהדר אינו בוחר את השיטה לביצוע עד לזמן הריצה. בכדי לאלץ את המהדר לקרוא לגירסה הנכונה של השיטה, שהמימוש שלה נעשה באובייקט שיש לו ייחוס משתמע למחלקת הבסיס, משתמשים בשתי מילות מפתח חדשות: **virtual** ו-**override**. יש להשתמש ב-virtual עבור השיטה של מחלקת הבסיס, ולהשתמש ב-override עבור מימוש השיטה במחלקה הנגזרת. הנה שוב הדוגמה, והפעם היא מתפקדת כהלכה!

```
using System;

class Employee
{
   public Employee(string name)
   {
      this.Name = name;
   }

   protected string Name;
   public string name

   {
      get
      {
         return this.Name;
      }
   }

   virtual public void CalculatePay()
   {
      Console.WriteLine("Employee.CalculatePay called for {0}",
                        name);
   }
}

class ContractEmployee : Employee
{
   public ContractEmployee(string name)
   : base(name)
   {
   }
   override public void CalculatePay()
   {
      Console.WriteLine("ContractEmployee.CalculatePay called for {0}",
                        name);
   }
}

class SalariedEmployee : Employee
{
   public SalariedEmployee (string name)
   : base(name)
   {
   }
```

```
    override public void CalculatePay()
    {
        Console.WriteLine("SalariedEmployee.CalculatePay called for {0}",
                          name);
    }
}

class Poly3App
{
    protected Employee[] employees;
    public void LoadEmployees()
    {
        // Simulating loading from a database.
        employees = new Employee[2];

        employees[0] = new ContractEmployee("Kate Dresen");
        employees[1] = new SalariedEmployee("Megan Sherman");
    }

    public void DoPayroll()
    {
        foreach(Employee emp in employees)
        {
            emp.CalculatePay();
        }
    }

    public static void Main()
    {
        Poly3App poly3 = new Poly3App();
        poly3.LoadEmployees();
        poly3.DoPayroll();
    }
}
```

לפני הרצת היישום, הבה נעיף מבט בקוד IL הנוצר. כעת נשים לב שהשורה IL_0014 משתמשת בהוראה callvirt של MSIL, האומרת למהדר שהשיטה המדויקת שיש לקרוא לא תהיה ידועה עד לזמן הריצה, כי הדבר תלוי באובייקט הנגזר שמשתמשים בו:

```
.method public hidebysig instance void DoPayroll() il managed
{
  // Code size       34 (0x22)
  .maxstack  2
  .locals (class Employee V_0,
           class Employee[] V_1,
           int32 V_2,
           int32 V_3)
```

```
  IL_0000:  ldarg.0
  IL_0001:  ldfld      class Employee[] Poly3App::employees
  IL_0006:  stloc.1
  IL_0007:  ldloc.1
  IL_0008:  ldlen
  IL_0009:  conv.i4
  IL_000a:  stloc.2
  IL_000b:  ldc.i4.0
  IL_000c:  stloc.3
  IL_000d:  br.s       IL_001d
  IL_000f:  ldloc.1
  IL_0010:  ldloc.3
  IL_0011:  ldelem.ref
  IL_0012:  stloc.0
  IL_0013:  ldloc.0
  IL_0014:  callvirt   instance void Employee::CalculatePay()
  IL_0019:  ldloc.3
  IL_001a:  ldc.i4.1
  IL_001b:  add
  IL_001c:  stloc.3
  IL_001d:  ldloc.3
  IL_001e:  ldloc.2
  IL_001f:  blt.s      IL_000f
  IL_0021:  ret
} // end of method Poly3App::DoPayroll
```

ביצוע הקוד בנקודה זו אמור לתת את התוצאות האלו:

```
c:\>Poly3App
ContractEmployee.CalculatePay called for Kate Dresen
SalariedEmployee.CalculatePay called for Megan Sherman
```

הערה

לא ניתן להצהיר על שיטות וירטואליות כ-private, מכיון שעל פי ההגדרה הן לא ייראו במחלקות הנגזרות.

# Static Method

**שיטה סטטית** (static method) קיימת במחלקה כחלק ממנה, ולא במופע מסוים של המחלקה. כמו עם חברים סטטיים אחרים, יתרון שיטות אלו בכך שהן נפרדות ממופע מסוים של מחלקה, אך אינן "מזהמות" את המרחב הגלובלי של היישום ומבלי להיות מנוגדות לעקרונות של תכנון מוכוון-עצמים בכך שאינן שייכות למחלקה כלשהי. להדגמה, אשתמש בדוגמת API של מסד נתונים שכתבתי ב-#C. בהיררכיית המחלקות שלי היתה מחלקה בשם SQLServerDb אשר כללה את היכולות הבסיסיות NURD (new, update, read ו-delete), וגם כללה יכולת נוספת לתיקון מסד הנתונים.

השיטה Repair של המחלקה, אשר יועדה לתיקון מסד הנתונים, לא צריכה היתה לפתוח את מסד הנתונים עצמו. למעשה, פונקציית ODBC שבה השתמשתי (SQLConfigDataSource) חייבה שמסד הנתונים יהיה סגור עבור פעולה זו. עם זאת, בנאי המחלקה SQLServerDb פתח מסד נתונים ששמו צוין כארגומנט. לפיכך, שיטה סטטית היתה פתרון מושלם. יכולתי להגדיר שיטה במחלקה SQLServerDb אשר היתה שייכת אליה ועם זאת לא הייתי צריך לעבור דרך בנאי המחלקה. היתרון בצד הלקוח היה ברור, כי גם הוא לא צריך היה לממש את המחלקה SQLServerDb. בדוגמה הבאה נקראת שיטה סטטית RepairDatabase מתוך השיטה Main. שים לב שאין אנו יוצרים מופע של SQLServerDb לצורך זה:

```
using System;

class SQLServerDb
{
   // Bunch of other nonsalient members.
   public static void RepairDatabase()
   {
      Console.WriteLine("repairing database...");
   }
}

class StaticMethod1App
{
   public static void Main()
   {
      SQLServerDb.RepairDatabase();
   }
}
```

הגדרת שיטה סטטית מחייבת שימוש במילת המפתח static. לאחר מכן, המשתמש מפעיל את התחביר Class.Method בכדי לקרוא לשיטה. שים לב שתחביר זה דרוש גם אם למשתמש יש ייחוס למופע של המחלקה. בכדי להמחיש נקודה זו, הידור הקוד הבא ייכשל:

```
// This code will fail to compile.
using System;

class SQLServerDb
{
   // Bunch of other nonsalient members.
   public static void RepairDatabase()
   {
      Console.WriteLine("repairing database...");
   }
}
```

```
class StaticMethod2App
{
   public static void Main()
   {
      SQLServerDb db = new SQLServerDb();
      db.RepairDatabase();
   }
}
```

## גישה לחברי מחלקה

עניין נוסף שיש לדון בו בהקשר של שיטות סטטיות הוא הכלל המסדיר את הגישה לחברי מחלקה מתוך שיטות סטטיות. כפי שניתן לשער, לשיטה סטטית יש גישה לכל חבר סטטי בתוך מחלקה, אך אין לה גישה לחבר של מופע. הקוד הבא מדגים זאת:

```
using System;

class SQLServerDb
{
   static string progressString1 = "repairing database...";
   string progressString2 = "repairing database...";

   public static void RepairDatabase()
   {
      Console.WriteLine(progressString1);     // This will work.
      Console.WriteLine(progressString2);     // Fails compilation.
   }
}

class StaticMethod3App
{
   public static void Main()
   {
      SQLServerDb.RepairDatabase();
   }
}
```

# סיכום

שיטות מקנות למחלקות את תכונות ההתנהגות ומבצעות פעולות למעננו. השיטות ב-#C הן גמישות – מאפשרות החזרה של ערכים מרובים, העמסה ומספר פרמטרים משתנה. מילות המפתח ref ו-out מאפשרות לשיטה להחזיר יותר מערך אחד לשיטה הקוראת. העמסה מאפשרת לשיטות מרובות בעלות שם זהה לתפקד באופן שונה, כתלות בסוגים ו/או מספר הארגומנטים המועברים אליהן. שיטות יכולות לקבל מספר משתנה של פרמטרים. מילת המפתח params מאפשרת לממש שיטות כאשר מספר הארגומנטים אינו ידוע עד לזמן ריצה. שיטות וירטואליות מאפשרות לשלוט בדרך שבה משנים את השיטות במחלקות נגזרות. לבסוף, מילת המפתח static מאפשרת להגדיר שיטות שקיימות כחלק ממחלקה, ולא כחלק מאובייקט.

# פרק 7 מאפיינים (Properties), מערכים (Arrays) וסדרנים (Indexers)

עד כה תיארתי את הסוגים הבסיסיים שבהם תומכת #C והסברתי איך צריך להצהיר ולהשתמש בהם במחלקות וביישומים. פרק זה חורג מהדפוס של הצגת תכונה אחת עיקרית של השפה בכל פרק. כאן תלמד על מאפיינים, מערכים וסדרנים גם יחד, מכיון שתכונות אלו של השפה קשורות זו לזו. הן מאפשרות למפתח #C להרחיב את מבנה המחלקה הבסיסי, כדי לחשוף ממשק טבעי ואינטואיטיבי יותר של חברי המחלקה.

## Properties as Smart Fields

תמיד טוב לתכנן מחלקות אשר לא רק מסתירות את מימוש שיטותיהן, אלא גם מונעות גישה ישירה של חבר לשדות של מחלקה. באמצעות **שיטות גישה** (accessor methods) שתפקידן לקרוא או לכתוב את הערכים של שדות אלה, ניתן להבטיח טיפול תקין בשדה – כלומר, בהתאם לחוקים של מרחב הבעיה המסוים – וביצוע פעולות עיבוד נוספות על פי הנדרש.

לשם דוגמה, נתייחס למחלקה המייצגת כתובת ומכילה שדות של מיקוד (ZIP code) ועיר. אם הלקוח משנה את השדה Address.ZipCode נרצה לתת תוקף לערך החדש על ידי בדיקה מול מאגר נתונים כלשהו, ולקבוע את ערך השדה Address.City באופן אוטומטי על סמך המיקוד החדש. אם ללקוח היתה גישה ישירה ל-Address.ZipCode ציבורי, לא היתה דרך קלה לבצע את הבדיקה והעדכון האוטומטי האלה, כי כתיבת ערך ישירות לתוך משתנה חבר אינה מחייבת שיטה. לכן, במקום להעניק גישה ישירה לשדה Address.ZipCode, תכנון טוב יותר יהיה להגדיר את השדות Address.ZipCode ו-Address.City כ-protected ולספק שיטות גישה לכתיבה וקריאה של השדה Address.ZipCode. בדרך זו ניתן לצרף לשינוי זה קוד המבצע כל עבודה נוספת נדרשת.

את הדוגמה העוסקת ב- ZIP code ניתן לכתוב ב-#C כמוצג להלן. שים לב שהשדה הממשי עבור ZIpCode מוגדר protected ולכן אינו נגיש על ידי הלקוח, ושיטות הגישה GetZipCode ו-SetZipCode מוגדרות public:

```
class Address
  {
    protected string ZipCode;
    protected string City;
    public string GetZipCode()
    {
      return this.ZipCode;
    }
    public void SetZipCode(string ZipCode)
    {
      // Validate ZipCode against some datastore.
      this.ZipCode = ZipCode;
      // Update this.City based on validated ZIP code.
    }
  }
```

לאחר מכן הלקוח יכול לגשת לערך Address.ZipCode בדרך זו:

```
Address addr = new Address();
addr.SetZipCode("55555");
string zip = addr.GetZipCode();
```

## הגדרת מאפיינים ושימוש בהם

השימוש בשיטות גישה (accessor methods) פועל היטב וטכניקה זו מקובלת בקרב מתכנתים של מספר שפות מוכוונות-אובייקטים, כולל ++C ו-Java. עם זאת, #C מספקת מנגנון עשיר יותר הקרוי **מאפיינים** (properties) – בעל יכולות זהות לאלו של שיטות גישה והרבה יותר אלגנטי ונוח מהן בצד הלקוח. מאפיינים מאפשרים לכתוב יישום לקוח, המאפשר גישה לשדות של מחלקה כאילו היו שדות ציבוריים מבלי לדעת אם קיימת שיטת גישה (accessor methods)ד.

מאפיין של #C מורכב מהצהרה על שדה ושיטות גישה המשמשות לעדכון הערך של השדה. שיטות גישה אלו הן **שיטה קוראת** (getter) ו**שיטה כותבת** (setter), אשר קוראות את ערך השדה, או כותבות ערך בהתאמה. הנה שוב הדוגמה הקודמת, שבה השתמשתי הפעם במאפייני #C:

```
class Address
   {
      protected string city;
      protected string zipCode;

      public string ZipCode
      {
         get
         {
            return zipCode;
         }
         set
         {
            // Validate value against some datastore.
            zipCode = value;
            // Update city based on validated zipCode.
         }
      }
   }
```

שים לב שיצרתי שדה בשם Address.zipCode ומאפיין בשם Address.ZipCode (שים לב לגודל האות Z). במבט ראשון הדבר עשוי לבלבל את מי שסבור ש-Address.ZipCode הוא השדה הממשי ומתפלא מדוע יש להגדירו פעמיים. ובכן, יש להבדיל בין שדה למאפיין, והבה נזכור ש-Address.ZipCode הוא מאפיין (property). זהו אמצעי כללי להגדרת שיטות גישה (accessor method) לחבר מחלקה, המאפשר שימוש בתחביר של object.field האינטואיטיבי יותר. אם בדוגמה זו הייתי משמיט את השדה Address.zipCode ומשנה את המשפט בשיטה הכותבת מ- zipCode = value ל- ZipCode = value, הייתי גורם ללולאת קריאה אינסופית של השיטה הכותבת. שים לב לכך שהשיטה הכותבת אינה מקבלת ארגומנטים. הערך המועבר מוצב אוטומטית במשתנה שנקרא value הנגיש בתוך שיטה זו בלבד (מייד תראה ב-MSIL איך קורה הקסם הזה).

לאחר הגדרת המאפיין Address.ZipCode, הבה נראה את השינויים המתחייבים בקוד הלקוח:

```
Address addr = new Address();

addr.ZipCode = "55555";

string zip = addr.ZipCode;
```

ניתן לראות שגישת הלקוח לשדות היא אינטואיטיבית: אין צורך לנחש או לעיין בתיעוד (ולא עלינו, גם בקוד עצמו) בכדי לקבוע אם שדה הוא public, ואם לא, מהו שם שיטת הגישה.

## מה באמת עושה כאן המהדר

איך המהדר מאפשר לקרוא לשיטה עם התחביר הסטנדרטי object.field? מהיכן מגיע המשתנה value? בכדי לענות על שאלות אלו עלינו לעיין בקוד MSIL שנוצר על ידי המהדר. תחילה נבחן את השיטה הקוראת של המאפיין.

בדוגמה שלנו, הוגדרה השיטה הקוראת כך:

```
class Address
{
   protected string city;
   protected string zipCode;
   public string ZipCode
   {
      get
      {
         return zipCode;
      }
       ...
   }
       ...
}
```

כשתתבונן בקוד MSIL המתקבל משיטה זו, תראה שהמהדר יצר שיטת גישה בשם get_ZipCode:

```
.method public hidebysig specialname instance string
      get_ZipCode() cil managed
{
  // Code size       11 (0xb)
  .maxstack  1
  .locals ([0] string _Vb_t_$00000003$00000000)
  IL_0000:  ldarg.0
  IL_0001:  ldfld      string Address::zipCode
  IL_0006:  stloc.0
  IL_0007:  br.s       IL_0009
  IL_0009:  ldloc.0
  IL_000a:  ret
} // end of method Address::get_ZipCode
```

תוכל לדעת את שם שיטת הגישה, כי המהדר מוסיף לשם המאפיין את הקידומת get_ (עבור השיטה הקוראת) או set_ (עבור השיטה הכותבת). כתוצאה מכך, הקוד הבא מתורגם לקריאה ל-get_ZipCode:

```
String str = addr.ZipCode;            // this calls Address::get_ZipCode
```

מכאן אולי תתפתה לנסות בעצמך את הקריאה המפורשת הבאה לשיטת הגישה:

```
String str = addr.get_ZipCode;        // **ERROR " Won't compile
```

עם זאת, הידור הקוד ייכשל, מכיון שקריאה מפורשת לשיטה פנימית של MSIL אינה חוקית.

התשובה לשאלתנו – איך המהדר מאפשר להשתמש בתחביר הסטנדרטי object.field ולגרום לקריאה לשיטה – היא, שהמהדר למעשה יוצר עבורנו את השיטות הקוראת והכותבת המתאימות כאשר הוא מנתח את תחביר המאפיין ב-#C. לפיכך, במקרה של המאפיין Address.ZipCode המהדר יוצר קוד MSIL המכיל את השיטות get_ZipCode ו-set_ZipCode לפי הצורך.

כעת נבחן את הקוד שנוצר עבור השיטה הכותבת. במחלקה Address ראינו את הקוד הזה:

```
public string ZipCode
   {
       ...
       set
      {
         // Validate value against some datastore.
         zipCode = value;
        // Update city based on validated zipCode.
      }
   }
```

שים לב שבקטע הקוד אין הצהרה על משתנה בשם value. למרות זאת, אנו יכולים להשתמש בקוד לאחסון הערך המועבר על ידי הקוד הקורא ולקבוע את ערכו של השדה המוגן zipCode. כאשר מהדר #C יוצר את MSIL עבור השיטה הכותבת, הוא שותל משתנה זה כארגומנט בשיטה הנקראת set_ZipCode.

בקוד MSIL שנוצר, שיטה זו מקבלת ארגומנט של משתנה מסוג string:

```
.method public hidebysig specialname instance void
     set_ZipCode(string 'value') cil managed
{
  // Code size       8 (0x8)
  .maxstack  8
  IL_0000:  ldarg.0
  IL_0001:  ldarg.1
  IL_0002:  stfld      string Address::zipCode
  IL_0007:  ret
}       // end of method Address::set_ZipCode
```

למרות שאינך יכול לראות שיטה זו בקוד המקור של C#, כאשר אתה מספק למאפיין ZipCode ערך כמו addr.ZipCode("12345"), הוא מתורגם לקריאה של MSIL ל-Address::set_ZipCode("12345"). כמו שראינו בשיטה get_ZipCode, ניסיון לקרוא ישירות לשיטה זו יגרור הודעת שגיאה.

## מאפיינים לקריאה בלבד

בדוגמה הקודמת, המאפיין Address.ZipCode הינו מסוג קריאה/כתיבה, כי מוגדרות שיטה קוראת וגם שיטה כותבת. יש מקרים בהם אינך רוצה שהלקוח יוכל לשנות את הערך של שדה נתון, ולכן תיצור שדה לקריאה בלבד על ידי השמטת השיטה הכותבת. בכדי להמחיש מהו מאפיין לקריאה בלבד, נמנע מהלקוח להגדיר ערך לשדה Address.City, וכך המאפיין Address.ZipCode נשאר היחידי שניתן לקרוא וגם לשנות את ערכו:

```
class Address
  {
    protected string city;
    public string City
    {
      get
      {
        return city;
      }
    }

    protected string zipCode;
    public string ZipCode
    {
      get
      {
        return zipCode;
      }
      set
      {
        // Validate value against some datastore.
        zipCode = value;
        // Update city based on validated zipCode.
      }
    }
  }
```

## הורשת מאפיינים

כמו עם שיטות, ניתן להשתמש במציינים virtual, override או abstract עבור מאפיינים. כך יכולה מחלקה נגזרת לרשת ולדרוס מאפיינים כמו שנעשה לכל חבר אחר ממחלקת הבסיס. הנקודה החשובה כאן לציון היא שניתן להגדיר מציינים אלה ברמת המאפיין בלבד. במילים אחרות, במקרים שבהם יש גם שיטה קוראת וגם שיטה כותבת, אם אתה דורס את אחת מהן, אתה חייב לדרוס את שתיהן.

## שימוש מתקדם במאפיינים

עד כה ראינו שמאפיינים שימושיים מהסיבות הבאות:

- הם מספקים רמה של הפשטה עבור לקוחות.
- הם מספקים אמצעי כללי לגישה אל חברי מחלקה על ידי שימוש בתחביר object.field.
- הם מאפשרים למחלקה להבטיח שניתן לבצע כל עיבוד נוסף כאשר משנים או ניגשים לשדה מסוים.

הסיבה השלישית מביאה אותנו לשימוש מועיל נוסף במאפיינים, הנקרא **אתחול עצל** (lazy initialization). זו טכניקת **מיטוב** (optimization) שבה חלק מחברי מחלקה אינם מאותחלים עד לרגע שבו זקוקים להם. אתחול עצל מועיל כאשר יש מחלקה המכילה חברים שפונים אליהם לעיתים רחוקות ואשר אתחולם נמשך זמן רב או צורך משאבים רבים. דוגמאות לכך הם מצבים שבהם יש לקרוא את הנתונים ממסד נתונים או מרשת רחוקה. מכיון שידוע שפנייה לחברים אלה אינה נעשית לעיתים קרובות ואתחולם יקר, ניתן להשהות את אתחולם עד שהשיטות הקוראות שלהם נקראות. לשם המחשה, נניח שקיים יישום מלאי שמופעל על ידי נציגי מכירות במחשבים נישאים בכדי לקלוט הזמנות של לקוחות, והם משתמשים בו מדי פעם בכדי לבדוק רמות מלאי. באמצעות מאפיינים ניתן לאפשר אתחול של המחלקה הרלוונטית מבלי לקרוא את רשומות המלאי, כמוצג בקוד שלהלן. השיטה הקוראת תפנה אל מסד הנתונים המרוחק רק כאשר נציג מכירות ירצה לברר מצב מלאי של פריט מסוים.

```
class Sku
  {
    protected double onHand;

    public string OnHand
    {
      get
      {
        // Read from central db and set onHand value.
        return onHand;
      }
    }
  }
```

כפי שראית עד כה, המאפיינים מאפשרים לספק שיטות גישה לשדות וממשק כללי ואינטואיטיבי עבור הלקוחות. משום כך, לעיתים מתייחסים אל מאפיינים כאל **שדות חכמים** (smart fields). כעת נתקדם צעד אחד ונלמד להגדיר ולהשתמש במערכים. כמו כן נראה איך משתמשים במאפיינים יחד עם מערכים בצורה של סדרנים (indexers).

# מערכים

הדוגמאות שהצגתי עד כה הראו איך להגדיר משתנים בעלי ערכים סופיים וידועים מראש. עם זאת, ברוב היישומים המעשיים אינך יודע מהו המספר המדויק של אובייקטים שתהיה זקוק לו עד לזמן ריצה. לדוגמה, אם אתה מפתח יישום של עורך טקסט וברצונך לעקוב אחר הפקדים שנוספים לתיבת דו-שיח, המספר המדויק של פקדים שהעורך יציג אינו ידוע מראש. עם זאת, תוכל להשתמש במערך בכדי לאחסן ולעקוב אחר קבוצות אובייקטים שמוקצים באופן דינמי – במקרה זה, הפקדים שבעורך.

ב-#C מערכים הם אובייקטים והמחלקה System.Array מוגדרת כמחלקת הבסיס שלהם. לפיכך, למרות שהתחביר של הגדרת מערך נראה דומה לזה הקיים ב- ++C או ב-Java, אתה יוצר מופע למעשה ממחלקה של NET. כלומר, כל מערך שמצהירים עליו מכיל את כל אותם החברים שעברו בהורשה מ- System.Array. כעת נלמד איך להצהיר וליצור מופע ממערכים, איך לעבוד עם סוגי המערכים השונים ואיך לבצע איטרציה על האלמנטים של מערך. כמו כן נציג כמה מהמאפיינים והשיטות השכיחים בשימוש של המחלקה System.Array.

## הצהרה על מערכים

הצהרה על מערך ב-#C נעשית על ידי הצבת סוגריים מרובעים ריקים בין שם הסוג לבין שם המשתנה של המערך, כמו בדוגמה זו:

```
int[] numbers;
```

שים לב שתחביר זה שונה במעט מזה המוכר לך מ- ++C, שבו הסוגריים המרובעים מצוינים לאחר שם המשתנה. מערכים הם מבוססי-מחלקה, ולכן רבים מאותם החוקים שחלים על הצהרת מחלקה מתאימים גם להם. לדוגמה, כאשר אתה מצהיר על מערך אינך מממש אותו מיד. כמו עם מחלקה, עליך ליצור מופע של המערך בכדי שהוא יתממש כמערך לכל דבר, ויכיל ערכים או אלמנטים אחרים. בדוגמה הבאה, אני מצהיר וגם יוצר באותו זמן מופע של מערך:

```
// Declares and instantiates a single-
// dimensional array of 6 integers.
int[] numbers = new int[6];
```

עם זאת, כאשר מצהירים על מערך כחבר של מחלקה, יש להצהיר עליו וליצור מופע שלו. שתי הדרכים המוצגות חוקיות ועובדות:

```
class YourClass
{
   ...
   int[] numbers;
   ...
   void SomeInitMethod()
   {
      ...
      numbers = new int[6];
      ...
   }
}
```

```
class YourClass
{
  ...
  int[] numbers = new int[6];
  ...
  void SomeInitMethod()
  {
    ...
  }
}
```

## דוגמה של מערך חד-מימדי

הנה דוגמה של הצהרה על מערך חד-מימדי כחבר מחלקה ויצירת מופע והצבת האלמנטים שלו בבנאי. השלב הבא הוא שימוש בלולאה for בכדי לבצע איטרציה על המערך ולהדפיס כל אלמנט שלו:

```
using System;

class SingleDimArrayApp
{
   protected int[] numbers;

   SingleDimArrayApp()
   {
      numbers = new int[6];
      for (int i = 0; i < 6; i++)
      {
         numbers[i] = i * i;
      }
   }

   protected void PrintArray()
   {
      for (int i = 0; i < numbers.Length; i++)
      {
         Console.WriteLine("numbers[{0}]={1}", i, numbers[i]);
      }
   }
```

```
    public static void Main()
    {
        SingleDimArrayApp app = new SingleDimArrayApp();
        app.PrintArray();
    }
}
```

ביצוע הדוגמה מפיק את הפלט הבא:

```
numbers[0]=0
numbers[1]=1
numbers[2]=4
numbers[3]=9
numbers[4]=16
numbers[5]=25
```

בדוגמה זו, השיטה SingleDimArray.PrintArray משתמשת במאפיין Length של System.Array בכדי לקבוע את מספר האלמנטים במערך. אמנם כאן זה לא ברור, מכיון שיש לנו מערך חד-מימדי, אך המאפיין Length מחזיר למעשה את מספר כל האלמנטים בכל המימדים של מערך. לפיכך, במקרה של מערך דו-מימדי בגודל 5 על 4, המאפיין Length יחזיר את הערך 20. בסעיף הבא נסתכל על מערכים רב-מימדיים ונלמד איך קובעים מהו הגבול העליון של מימד מסוים במערך.

## מערכים רב-מימדיים

בנוסף למערכים חד-מימדיים, #C תומכת בהצהרה על מערכים רב-מימדיים שבהם כל מימד של המערך מופרד באמצעות פסיק. בדוגמה זו הצהרה על מערך תלת-מימדי מסוג double:

```
double[,,] numbers;
```

בכדי לקבוע במהירות את מספר המימדים במערך מוצהר, יש למנות את מספר הפסיקים ולהוסיף 1. בדוגמה זו היו 2 פסיקים, נוסיף 1 והרי לנו הצהרה על מערך תלת-מימדי.

בדוגמה הבאה מוצג מערך דו-מימדי של נתוני מכירות המייצג את נתוני המכירות של השנה הנוכחית עד עתה, ואת סך המכירות של שנה קודמת עבור אותו חלון זמן. שים לב במיוחד לתחביר המשמש ליצירת מופע של המערך (בבנאי MultiDimArrayApp):

```
using System;

class MultiDimArrayApp
{
    protected int currentMonth;
    protected double[,] sales;
```

```
    MultiDimArrayApp()

    {
        currentMonth=10;

        sales = new double[2, currentMonth];
        for (int i = 0; i < sales.GetLength(0); i++)
        {
            for (int j=0; j < sales.GetLength(1); j++)
            {
                sales[i,j] = (i * 100) + j;
            }
        }
    }

    protected void PrintSales()
    {
        for (int i = 0; i < sales.GetLength(0); i++)
        {
            for (int j=0; j < sales.GetLength(1); j++)
            {
                Console.WriteLine("[{0}][{1}]={2}", i, j, sales[i,j]);
            }
        }
    }

    public static void Main()
    {
        MultiDimArrayApp app = new MultiDimArrayApp();
        app.PrintSales();
    }
}
```

ביצוע הדוגמה MultiDimArrayApp מפיק את הפלט הבא:

```
[0][0]=0
[0][1]=1
[0][2]=2
[0][3]=3
[0][4]=4
[0][5]=5
[0][6]=6
[0][7]=7
[0][8]=8
[0][9]=9
[1][0]=100
```

```
[1][1]=101
[1][2]=102
[1][3]=103
[1][4]=104
[1][5]=105
[1][6]=106
[1][7]=107
[1][8]=108
[1][9]=109
```

בדוגמת המערך החד-מימדי ציינתי שהמאפיין Length יחזיר את מספר הפריטים (או האיברים) במערך, ולכן בדוגמה זו הערך המוחזר יהיה 20 (2x10). בשיטה MultiDimArray.PrintSales השתמשתי בשיטה Array.GetLength כדי לקבוע את האורך או הגבול העליון של כל מימד במערך.

## אחזור מספר המימדים

לאחר שראינו באיזו קלות ניתן לבצע איטרציה באופן דינמי על מערך חד או דו-מימדי, גם נרצה לדעת מהו מספר המימדים במערך. מספר המימדים במערך נקרא **דרג** המערך (rank), וניתן לקרוא אותו על ידי המאפיין Array.Rank. הנה דוגמה למציאת דרג של מערכים שונים:

```
using System;

class RankArrayApp
{
   int[] singleD;
   int[,] doubleD;
   int[,,] tripleD;

   protected RankArrayApp()
   {
      singleD = new int[6];
      doubleD = new int[6,7];
      tripleD = new int[6,7,8];
   }

   protected void PrintRanks()
   {
      Console.WriteLine("singleD Rank = {0}", singleD.Rank);
      Console.WriteLine("doubleD Rank = {0}", doubleD.Rank);
      Console.WriteLine("tripleD Rank = {0}", tripleD.Rank);
   }
```

```
    public static void Main()
    {
        RankArrayApp app = new RankArrayApp();
        app.PrintRanks();
    }
}
```

כצפוי, היישום RankArrayApp מפיק את הפלט הבא:

```
singleD Rank = 1
doubleD Rank = 2
tripleD Rank = 3
```

## מערכים משוננים

מושג אחרון שנתייחס אליו בהקשר למערכים הוא **מערך משונן** (jagged array), שהוא למעשה מערך של מערכים. הנה דוגמה להגדרת מערך המכיל מערכים מסוג integer:

```
int[][] jaggedArray;
```

אתה עשוי להשתמש במערך משונן בפיתוח יישום של עורך טקסט. בעורך כזה תרצה לשמור את האובייקט המייצג כל פקד שנוצר על ידי המשתמש במערך. נניח שיש שלושה לחצנים ושתי תיבות רשימה נפתחות המאוחסנים במערכים מתאימים ונפרדים. הצהרה על מערך משונן מאפשרת להחזיק מערך "אב" עבור מערכים אלה, כך שהתוכנית תוכל בקלות לבצע איטרציה על הפקדים כאשר יש צורך בכך, כמוצג כאן:

```
using System;

class Control
{
    virtual public void SayHi()
    {
        Console.WriteLine("base control class");
    }
}

class Button : Control
{
    override public void SayHi()
    {
        Console.WriteLine("button control");
    }
}
```

```
class Combo : Control
{
   override public void SayHi()
   {
      Console.WriteLine("combobox control");
   }
}

class JaggedArrayApp
{
   public static void Main()
   {
   Control[][] controls;
   controls = new Control[2][];

   controls[0] = new Control[3];
   for (int i = 0; i < controls[0].Length; i++)
   {
      controls[0][i] = new Button();
   }

   controls[1] = new Control[2];
   for (int i = 0; i < controls[1].Length; i++)
   {
      controls[1][i] = new Combo();
   }

   for (int i = 0; i < controls.Length;i++)
   {
      for (int j=0;j< controls[i].Length;j++)
      {
         Control control = controls[i][j];
         control.SayHi();
      }
   }

   string str = Console.ReadLine();
   }
}
```

בדוגמה זו הגדרתי מחלקת בסיס (Control) ושתי מחלקות נגזרות (Button ו-Combo) והצהרתי על המערך המשונן כמערך של מערכים המכיל אובייקטים מסוג Control. בדרך זו ניתן לאחסן את הסוגים המסוימים במערך, ולדעת – באמצעות רב-צורתיות (polymorphism) – שכאשר מחלצים את האובייקטים מן המערך באמצעות אובייקט עם המרה כלפי מעלה (upcasted object), מקבלים את ההתנהגות לה מצפים.

# Indexers

בסעיף "מערכים" למדת להצהיר ולממש מערכים, לעבוד עם סוגי המערכים השונים, וגם לבצע איטרציה על אלמנטים של מערך. גם למדת איך לנצל את היתרון של כמה מהשיטות והמאפיינים היותר שכיחים של סוגי המערכים המונחים ביסודה של המחלקה System.Array. כעת נלמד איך תכונה של C# הנקראת **סדרנים** (indexers) מאפשרת להתייחס בדרך של תכנות אל אובייקטים כאילו היו מערכים.

מדוע נרצה לטפל באובייקט כאילו הוא מערך? כמו מרבית התכונות של שפת תכנות, יתרון השימוש בסדרנים (indexers) מסתכם בהענקת היכולת לכתוב יישומים בצורה אינטואיטיבית יותר. בסעיף הראשון של הפרק ראית איך מאפיינים של C# מספקים את היכולת להתייחס לשדות של מחלקה באמצעות התחביר הסטנדרטי class.field, המתורגם בסופו של דבר לשיטות הקוראת והכותבת (getter ו-setter). הפשטה זו משחררת את המתכנת שכותב יישום עבור המחלקה מן הצורך לקבוע אם השיטות קוראת/כותבת קיימות עבור השדה, וגם מן הצורך לדעת מהו הפורמט המדויק של שיטות אלו. באופן דומה, סדרנים מאפשרים ללקוח של המחלקה לגשת לאובייקט באמצעות אינדקס, כאילו האובייקט עצמו הוא מערך.

לדוגמה, נתייחס למחלקה של תיבת רשימה שצריכה לחשוף דרך כלשהי שבה המשתמש במחלקה זו יכול להכניס מחרוזות. אם אתה מכיר את SDK של Win32, אתה יודע שכדי להכניס מחרוזת לתוך חלון של תיבת רשימה יש לשלוח לחלון את ההודעה LB_ADDSTRING או LB_INSERTSTRING. כאשר מנגנון זה הופיע בסוף שנות השמונים, סברנו שאכן אנו מתכנתים מוכווני-עצמים. אחרי הכל, הרי שלחנו הודעות לאובייקט כפי שלימדו אותנו הספרים ההדורים שעסקו בתכנון וניתוח מוכוון-עצמים. אולם, עם התרבותן של שפות מסוג זה כמו C++ ו- Object Pascal, למדנו שניתן להשתמש באובייקטים ליצירת ממשקי תכנות אינטואיטיביים יותר למטרות כאלו. באמצעות C++ ו-MFC (Microsoft Foundation Classes), קיבלנו אוסף שלם של מחלקות שאיפשר להתייחס אל חלונות (כמו תיבת רשימה) כאל אובייקטים. מחלקות אלו חשפו פונקציות חברות אשר באופן בסיסי מספקות מעטפת דקה לשליחה וקבלה של הודעות אל רובד הבקרה של Microsoft Windows המונח ביסודן. במקרה של המחלקה CListBox (כלומר, העטיפה של MFC לפקד תיבת רשימה של Windows), קיבלנו את הפונקציות החברות AddString ו-InsertString עבור המשימות שהתבצעו בעבר באמצעות שליחת ההודעות LB_ADDSTRING ו-LB_INSERTSTRING.

עם זאת, בכדי לעזור בפיתוח שפה שתהיה האינטואיטיבית והטובה ביותר, צוות התכנון של שפת C# הסתכל על כל זה ותהה "מדוע שלא תהיה יכולת להתייחס לאובייקט שמיסודו הוא מערך, כמו אל מערך?" האם תיבת רשימה אינה למעשה מערך של מחרוזות עם פונקציונליות נוספת להצגת הרשימה ומיונה? כך נולד הרעיון להפעיל סדרנים (indexers).

## הגדרת סדרנים

לעיתים מתייחסים למאפיינים כאל "שדות חכמים", ולכן יש היגיון בכך שמאפיינים וסדרנים יהיו שותפים לאותו התחביר. אכן, הגדרת סדרנים דומה מאוד להגדרת מאפיינים, עם שני הבדלים עיקריים: הראשון, הסדרן מקבל ארגומנט **index**. השני, מכיון שהמחלקה עצמה משמשת כמערך, מילת המפתח **this** מציינת את שם הסדרן. הנה דוגמת מימוש של סדרן:

```
class MyClass
{
   public object this [int idx]
   {
      get
      {
         // Return desired data.
      }
      set
      {
         // Set desired data.
      }
   }
   ...
}
```

טרם הצגתי דוגמה מושלמת הממחישה את התחביר של סדרנים, כי המימוש הפנימי הממשי להגדרת הנתונים, וקריאה וכתיבה שלהם אינו רלוונטי לסדרנים. זכור כי מבלי להתחשב באופן שבו אתה שומר את הנתונים מבחינה פנימית (כלומר, מערך, אוסף וכן הלאה), סדרנים הם רק אמצעי תכנות בעת יצירת מופע ממחלקה, כדי לכתוב קוד בדומה לזה:

```
MyClass cls = new MyClass();
cls[0] = someObject;
Console.WriteLine("{0}", cls[0]);
```

מה שאתה עושה בתוך הסדרן הוא עניינך, כל זמן שמי שמפעיל את המחלקה מקבל את התוצאות שהוא מצפה להן בעת גישה אל אובייקט כמו אל מערך.

## דוגמה לסדרן

הבה נסתכל על מספר מקומות שבהם יש היגיון רב לשימוש בסדרנים. נתחיל עם דוגמת תיבת הרשימה הקודמת. כפי שציינתי, תיבת רשימה מבחינה מושגית היא רשימה, או מערך של מחרוזות טקסט המיועדות להצגה. בדוגמה הבאה הצהרתי על מחלקה בשם MyListBox, המכילה סדרן לקריאה וכתיבה של ערכי טקסט באמצעות אובייקט ArrayList (המחלקה ArrayList של NET משמשת לאחסון אוסף של אובייקטים).

```
using System;
using System.Collections;

class MyListBox

{
   protected ArrayList data = new ArrayList();

   public object this[int idx]
   {
      get
      {
         if (idx > -1 && idx < data.Count)
         {
            return (data[idx]);
         }
         else
         {
            // Possibly throw an exception here.
            return null;
         }
      }
      set
      {
         if (idx > -1 && idx < data.Count)
         {
            data[idx] = value;
         }
         else if (idx == data.Count)
         {
            data.Add(value);
         }
         else
         {
            // Possibly throw an exception here.
         }
      }
   }
}
```

```
class Indexers1App
{
   public static void Main()
   {
      MyListBox lbx = new MyListBox();
      lbx[0] = "foo";
      lbx[1] = "bar";
      lbx[2] = "baz";
      Console.WriteLine("{0} {1} {2}",
                        lbx[0], lbx[1], lbx[2]);
   }
}
```

בדוגמה זו כללתי בדיקת שגיאות של חריגה מגבולות עבור האינדקסים של הנתונים. מבחינה טכנית אין זה קשור לסדרנים כי סדרנים קשורים רק לאופן שבו לקוח של המחלקה יכול להשתמש באובייקט כמערך, ואין להם ולא כלום עם הייצוג הפנימי של הנתונים. עם זאת, כאשר לומדים תכונה חדשה של שפה, יהיה זה מועיל לראות שימוש מעשי בתכונה ולא רק להתייחס לתחביר שלה. לכן, גם בשיטה הקוראת וגם בשיטה הכותבת של הסדרן אני בודק את חוקיות ערכו של האינדקס המועבר ביחד עם הנתונים המאוחסנים בחבר ArrayList של המחלקה. באופן אישי אולי אעדיף לעורר חריגים במקרים שבהם לא ניתן לפרש את ערך האינדקס המועבר. עם זאת, מדובר בבחירה אישית – דרך הטיפול שלך בשגיאות עשויה להיות שונה. בכל מקרה, צריך לציין ללקוח מצב שגיאה כאשר מועבר אינדקס שאינו חוקי.

## קווים מנחים לתכנון

סדרנים הם דוגמה לתוספת עדינה אך רבת עוצמה של צוות התכנון של שפת #C, העוזרת לנו להפוך ליצרניים יותר במאמצי הפיתוח שלנו. עם זאת, כמו לכל תכונה של כל שפה, לסדרנים יש מקום משלהם. יש להשתמש בהם במקום שבו יהיה זה אינטואיטיבי לגשת אל אובייקט כאילו היה מערך. לשם דוגמה נתייחס למקרה של הפקת חשבוניות. סביר להניח שליישום כזה יש מחלקת Invoice המגדירה מערך חבר של אובייקטים מסוג InvoiceDetail (שורת פרטי חשבונית). במקרה כזה, יהיה זה אינטואיטיבי שהמשתמש ייגש לשורות פירוט אלו באמצעות התחביר הבא:

```
InvoiceDetail detail = invoice[2]; // Retrieves the 3rd detail line.
```

עם זאת, לא יהיה זה אינטואיטיבי להתקדם עוד צעד ולנסות להפוך את כל החברים InvoiceDetail למערך שיהיה נגיש דרך סדרן. כפי שאתה רואה, השורה הראשונה מובנת בקלות רבה יותר מהשנייה:

```
TermCode terms = invoice.Terms; // Property accessor to Terms member.
TermCode terms = invoice[3];    // A solution in search of a problem.
```

בעניין זה תקפה המימרה שגם אם יכולים לעשות משהו אין זה אומר שחייבים לעשות אותו. מבחינה מעשית, עליך לחשוב איך מימוש של תכונה חדשה כלשהי עומד להשפיע על לקוחות המחלקה שלך, ומחשבה זו צריכה להנחות אותך כאשר אתה צריך להחליט אם יישום התכונה יקל על השימוש במחלקה.

# סיכום

מאפיינים של #C מורכבים מהצהרה על שדה ושיטות גישה. הם מאפשרים גישה חכמה לשדות של מחלקה, כך שמתכנת הכותב מימוש של המחלקה אינו צריך לנסות ולקבוע אם (ואיך) נוצרה שיטת גישה עבור השדה. מצהירים על מערכים ב-#C על ידי הצבת סוגריים מרובעים ריקים **בין** הסוג ושם המשתנה, תחביר השונה מעט מזה המשמש ב- ++C. מערכים של #C יכולים להיות חד-מימדיים, רב-מימדיים או משוננים. ניתן לגשת לאובייקטים ב-#C כמו למערכים על ידי שימוש בסדרנים (indexers). סדרנים מאפשרים למתכנתים לעבוד עם אובייקטים רבים מאותו סוג ולעקוב אחריהם ביתר קלות.

# פרק 8 תכונות - Attributes

רוב שפות התכנות מתוכננות עם חשיבה על קבוצה נתונה של יכולות. לדוגמה, כאשר מפתחים מהדר, חושבים כיצד ייבנה יישום שנכתב בשפה החדשה, איך קוד יקרא לקוד אחר, איך תיארז הפונקציונליות ועוד סוגיות אחרות שיהפכו את השפה לאמצעי מועיל לפיתוח תוכנה. רוב הרעיונות של מתכנן מהדר הם סטטיים. לדוגמה, ב-#C מגדירים מחלקה על ידי כתיבת המציין class לפני שם המחלקה. כדי לציין גזירה צריך להוסיף נקודתיים לאחר שם המחלקה ולרשום בהמשך את שם מחלקת הבסיס. זו דוגמה להחלטה, אשר מרגע שנעשתה על ידי מפתח השפה לא ניתן לשנותה.

מפתחי המהדרים עושים את עבודתם כנדרש על פי תכתיבי השפה. עם זאת, אפילו הם אינם יכולים לחזות מראש את כל ההתפתחויות העתידיות בתעשייה וכיצד אלו ישנו את האופן שבו מתכנתים ירצו לממש את סוגי הנתונים שלהם בשפה נתונה. לדוגמה, איך יוצרים קשר בין מחלקה ב-++C וכתובת URL המכילה תיעוד עבור מחלקה זו? או, איך אתה משייך חברים מסוימים של מחלקת ++C לשדות XML להעברת מידע בין עסקים? מכיון ש-++C פותחה שנים רבות לפני האינטרנט והפרוטוקולים שלה (כמו XML, למשל), אין זה פשוט לבצע שתי משימות אלו.

עד כה, הפתרונות לבעיות מסוג זה היו כרוכים באחסון נתונים בקובץ נפרד (DEF, IDL וכן הלאה) אשר לאחר מכן היה מקושר בצורה רופפת עם הסוג או החבר הנידון. גישה זו נקראה "פתרון מנותק" ("disconnected solution"), כי המהדר אינו יודע על הקובץ הנפרד או על הקשר המקודד בין המחלקה והקובץ. הבעיה המרכזית היא שהמחלקה כבר אינה "מתארת את עצמה" – כלומר, המשתמש אינו יכול עוד להסתכל על הגדרת המחלקה לכשעצמה ולדעת הכל אודותיה. אחד היתרונות של רכיב שמתאר את עצמו הוא שהמהדר וזמן הריצה יכולים להבטיח ציות לחוקים הקשורים ברכיב. בנוסף לכך, רכיב שמתאר את עצמו קל יותר לתחזוקה, מכיון שהמפתח יכול לראות במקום אחד את כל המידע המתייחס אליו.

כך נהג העולם במשך עשורים רבים של התפתחות המהדרים. מתכנני השפה ניסו לקבוע מה תרצה שהשפה תעשה ותכננו את המהדר עם יכולות אלו, אשר היו תקפות עד שפותחה גירסה חדשה של המהדר, או עד אשר הופיע מהדר חדש בשוק. כלומר, C# מציעה כעת מודל עבודה שונה, הנובע מההצגה של יכולת הנקראת **תכונות** (attributes).

# הצגת תכונות

תכונות מאפשרות לשבור את הכללים, או את נוהל העבודה המקובל. הן מספקות דרך לשייך מידע (כגון הערות או דברי הסבר) לסוגים מוגדרים של C#. ניתן להגדיר באמצעותם מידע של זמן תכנון (כגון מידע על תיעוד), מידע של זמן ריצה (כגון שם עמודה של שדה במסד נתונים), או גם מאפיינים התנהגותיים של זמן ריצה (כגון האם חבר נתון יכול ליטול חלק בתנועת עדכון נתונים). מספר האפשרויות אינו מוגבל, וניתן ליצור תכונות על בסיס כל מידע שרוצים. לשם כך קיים מנגנון סטנדרטי להגדרת התכונות עצמן ולאחזור מידע באשר לתכונות המצורפות לחבר או לסוג בזמן הריצה.

להמחשת אופן השימוש ביכולת חזקה זו, נתאר לעצמנו יישום המאחסן חלק מהמידע שלו ברישום המערכת (Windows Registry). סוגיית תכנון שעולה כאן היא ההחלטה היכן לאחסן את המידע על מפתח הרישום. ברוב סביבות הפיתוח נשמר מידע זה בדרך כלל בקובץ משאבים נלווה, בתוך קבועים או כחלק מהקוד של הקריאות ל-API של הרישום (קידוד קשיח). עם זאת, אנו רואים שגם כאן שומרים חלק אינטגרלי של המחלקה בנפרד מיתר ההגדרה שלה. השימוש בתכונות מאפשר "לצרף" מידע זה לחברי המחלקה, כך שהרכיב מתאר לגמרי את עצמו. הנה, כך זה נראה, בהנחה שכבר הגדרנו תכונה בשם RegistryKey במקום אחר:

```
class MyClass
{
   [RegistryKey(HKEY_CURRENT_USER, "foo")]
   public int Foo;
}
```

בכדי לצרף תכונה מוגדרת לסוג או לחבר של C#, מציינים לפני הסוג או החבר את נתוני התכונה בתוך סוגריים מרובעים. כאן צירפנו תכונה בשם RegistryKey לשדה MyClass.Foo. בזמן ריצה, כל שעלינו לעשות הוא לקרוא מהשדה את מידע מפתח הרישום ולהשתמש בערך זה בכדי לשמור את הנתונים ברישום.

# הגדרת תכונות

שים לב שהתחביר המשמש לצירוף תכונה לסוג או לחבר דומה במעט ליצירת מופע ממחלקה. זאת, מכיון שתכונה היא אכן מחלקה אשר נגזרת ממחלקת הבסיס System.Attribute.

הנה הקוד עבור התכונה RegistryKey שבדוגמה הקודמת:

```
public enum RegistryHives
{
   HKEY_CLASSES_ROOT,
   HKEY_CURRENT_USER,
   HKEY_LOCAL_MACHINE,
   HKEY_USERS,
   HKEY_CURRENT_CONFIG
}

public class RegistryKeyAttribute : Attribute
{
   public RegistryKeyAttribute(RegistryHives Hive, String ValueName)
   {
      this.Hive = Hive;
      this.ValueName = ValueName;
   }

   protected RegistryHives hive;
   public RegistryHives Hive
   {
      get { return hive; }
      set { hive = value; }
   }

   protected String valueName;
   public String ValueName
   {
      get { return valueName; }
      set { valueName = value; }
   }
}
```

כאן הוספתי enum (מנייה) עבור סוגי הרישום השונים, בנאי עבור מחלקת התכונה (עם פרמטרים של עץ משנה ושם ערך), ושני מאפיינים עבור עץ המשנה של הרישום ושם ערך. ניתן לעשות הרבה יותר כאשר מגדירים תכונות, אך כעת, לאחר שאנו כבר יודעים להגדיר ולצרף תכונות, נלמד איך לקרוא מידע של תכונות בזמן ריצה. כך נקבל דוגמה מושלמת שנוכל להשתמש בה. לאחר שנסיים, נעבור לכמה סוגיות מתקדמות יותר בהגדרת וצירוף תכונות.

הערה

שים לב בדוגמאות, שהמילה Attribute מצורפת לשם של כל מחלקת תכונה. עם זאת, כאשר נצרף את התכונה לסוג או חבר, לא נוסיף סיומת זו. זהו קיצור דרך הניתן חינם על ידי מתכנני שפת #C. כאשר המהדר מבחין שתכונה מצורפת לסוג או לחבר, הוא יחפש מחלקה נגזרת של System.Attribute עם שם התכונה המסוים. אם לא ניתן לאתר את המחלקה, המהדר יוסיף לשם התכונה את המילה Attribute ויחפש אותה. לכן, נוהג מקובל הוא להגדיר שם של כל מחלקת תכונה עם הסיומת Attribute ולאחר מכן להשמיט חלק זה של השם.

# קריאת מידע של תכונות

אנו יודעים איך להגדיר תכונה על ידי גזירתה מ-System.Attribute וגם יודעים לצרפה לסוג או לחבר. כעת נרצה לדעת איך להשתמש בתכונות בקוד. במילים אחרות, איך ניתן לקרוא מסוג או חבר את מידע התכונות (והפרמטרים שלו) שהוצמדו אליהם?

בכדי לקרוא תכונות המצורפות לסוג או חבר עלינו להשתמש ב"שיקוף". **שיקוף** (reflection) הוא נושא מתקדם הנסקר בפרק 16, ולכן נדון בו כאן רק במידה הדרושה בכדי להסביר מה נחוץ לקריאת מידע של תכונה בזמן ריצה. אם ברצונך ללמוד יותר על שיקופים, עיין בפרק 16.

שיקוף הוא יכולת לקבוע באופן דינמי בזמן ריצה את מפרט הסוגים של יישום. לדוגמה, ניתן להשתמש בממשקי התכנות (APIs) מסוג Reflection של סביבת NET בכדי לבצע איטרציה על מטא-נתונים עבור מכלל קוד שלם, וליצור רשימה של כל המחלקות, הסוגים והשיטות המוגדרים במכלל קוד זה. הבה נבחן מספר דוגמאות של תכונות ונלמד לקרוא אותן באמצעות שיקוף.

## תכונות של מחלקות

אופן קריאת תכונה תלוי בסוג החבר שממנו רוצים לקבל את המידע. נניח שרוצים להגדיר תכונה המגדירה את השרת המרוחק שבו יש ליצור אובייקט. ללא תכונות היינו צריכים לשמור מידע זה בקבוע, או בקובץ המשאבים של היישום. השימוש בתכונות מאפשר לצרף למחלקה מידע על סוג השרת המרוחק, בדומה לזה:

```
using System;

public enum RemoteServers
{
   JEANVALJEAN,
   JAVERT,
   COSETTE
}
```

```
public class RemoteObjectAttribute : Attribute
{
   public RemoteObjectAttribute(RemoteServers Server)
   {
      this.server = Server;
   }

   protected RemoteServers server;
   public string Server
   {
      get
      {
         return RemoteServers.GetName(typeof(RemoteServers), this.server);
      }
   }
}

[RemoteObject(RemoteServers.COSETTE)]
class MyRemotableClass
{
}
```

בכדי לקבוע באיזה שרת ליצור את האובייקט, נשתמש בקוד כמו זה:

```
class ClassAttrApp
{
   public static void Main()
   {
      Type type = typeof(MyRemotableClass);
      foreach (Attribute attr in type.GetCustomAttributes())
      {
         RemoteObjectAttribute remoteAttr =
            attr as RemoteObjectAttribute;
         if (null != remoteAttr)
         {
            Console.WriteLine("Create this object on {0}.", remoteAttr.Server);
         }
      }
   }
}
```

הנה הפלט המתקבל מיישום זה:

Create this object on COSETTE.

מכיון שחלק מהקוד משותף לכל הגרסאות של דוגמה זו, נבחן כעת את השימוש בשיקוף ואיך הוא מחזיר את ערך התכונה בזמן ריצה.

בשורה הראשונה של השיטה Main נעשה שימוש באופרטור typeof:

```
Type type = typeof(MyRemotableClass);
```

אופרטור זה מקבל ארגומנט יחיד ומחזיר את האובייקט System.Type המשויך אליו. ברגע שמקבלים את האובייקט, אפשר להתחיל לקבל ממנו מידע.

שורת הקוד הבאה מחייבת הסבר בשני חלקים:

```
foreach (Attribute attr in type.GetCustomAttributes(true))
```

החלק הראשון הוא הקריאה לשיטה type.GetCustomAttributes. שיטה זו מחזירה מערך מסוג Attribute, אשר במקרה זה מכיל את כל התכונות המצורפות למחלקה MyRemotableClass. החלק השני הוא לולאת foreach, הניגשת לכל אחד מן האלמנטים של המערך המוחזר ומכניסה את ערכו לתוך המשתנה attr מסוג Attribute.

המשפט הבא משתמש באופרטור as בכדי לנסות להמיר את המשתנה attr לסוג RemoteObjectAttribute:

```
RemoteObjectAttribute remoteAttr =
      attr as RemoteObjectAttribute;
```

לאחר מכן אנו בודקים באמצעות האופרטור as את הערך null (ריק), המציין תוצאה של כישלון. אם הערך אינו null – כלומר, המשתנה remoteAttr מכיל תכונה חוקית המצורפת לסוג MyRemotableClass – אנו קוראים לאחד המאפיינים של RemoteObjectAttribute בכדי להדפיס את שם השרת:

```
if (null != remoteAttr)
{
   Console.WriteLine("Create this object on {0}",
                     remoteAttr.Server);
}
```

## תכונות של שיטות

לאחר שלמדנו איך לפעול עם תכונות של מחלקות, נסקור את השימוש בתכונות של שיטות. כאן הוקדש לדיון סעיף נפרד, כי קוד השיקוף הדרוש לקריאת תכונה של שיטה שונה מזה הנדרש לקריאת תכונה של מחלקה. בדוגמה זו, נגדיר שיטה בעלת התכונה transactionable:

```
using System;
using System.Reflection;

public class TransactionableAttribute : Attribute
{
   public TransactionableAttribute()
   {
   }
}

class TestClass
{
   [Transactionable]
   public void Foo()
   {}

   public void Bar()
   {}

   [Transactionable]
   public void Baz()
   {}
}

class MethodAttrApp
{
   public static void Main()
   {
      Type type = Type.GetType("TestClass");
      foreach(MethodInfo method in type.GetMethods())
      {
         foreach (Attribute attr in
               method.GetCustomAttributes())
```

```
        {
            if (attr is TransactionableAttribute)
            {
                Console.WriteLine("{0} is transactionable.", method.Name);
            }
        }
    }
  }
}
```

הנה הפלט המתקבל:

```
Foo is transactionable.
Baz is transactionable.
```

בדוגמה מסוימת זו, הנוכחות של TransactionableAttribute מספיקה לכשעצמה בכדי לדעת ששיטה בעלת תכונה כזו היא transactionable. לכן, התכונה מוגדרת עם בנאי ריק ללא פרמטרים וללא חברים אחרים.

בהמשך מוגדרת TestClass עם שלוש שיטות (Foo, Bar ו-Baz), אשר שתיים מהן (Foo ו-Baz) מוגדרות transactionable. שים לב שכאשר מצרפים תכונה עם בנאי נטול פרמטרים, אין צורך לכלול את הסוגריים הפותחים והסוגרים.

כעת הגענו לחלק המעניין. הבה נבחן מקרוב איך נוכל לקרוא מידע על תכונות של שיטת מחלקה. אנו מתחילים על ידי שימוש בשיטה הסטטית GetType של Type בכדי לקבל את אובייקט System.Type עבור המחלקה TestClass:

```
Type type = Type.GetType("TestClass");
```

לאחר מכן משתמשים בשיטה type.GetMethods לקבלת מערך אובייקטים מסוג MethodInfo. כל אחד מאובייקטים אלה מכיל את המידע על שיטה במחלקה TestClass. באמצעות לולאת foreach אנו ניגשים לכל אחת מהשיטות:

```
foreach(MethodInfo method in type.GetMethods())
```

כעת כשיש בידנו אובייקט MethodInfo, אנו יכולים להשתמש בשיטה GetCustomAttributes של MethodInfo בכדי לקבל את כל התכונות המותאמות עבור השיטה. שוב נשתמש בלולאת foreach בכדי לבצע איטרציה על המערך המוחזר של האובייקטים:

```
foreach (Attribute attr in method.GetCustomAttributes(true))
```

בנקודה זו של הקוד יש בידנו תכונה עבור שיטה. באמצעות האופרטור is, אנו בודקים אם התכונה היא TransactionableAttribute. אם כן, אנו מדפיסים את שם השיטה:

```
if (attr is TransactionableAttribute)
{
Console.WriteLine("{0} is transactionable.", method.Name);
}
```

## תכונות של שדות

בדוגמה האחרונה של קריאת מידע מחברים על התכונות המצורפות אליהם, נלמד לקרוא מידע משדות של מחלקה. נניח שיש מחלקה המכילה מספר שדות, שאת ערכיהם אנו רוצים לשמור ברישום המערכת. לשם כך ניתן להגדיר תכונה עם בנאי שמקבל כפרמטרים enum (מנייה) המייצג את עץ המשנה המתאים, ומחרוזת המייצגת את שם ערך הרישום (שם המפתח). בדרך זו ניתן יהיה לקרוא מן השדה בזמן ריצה מידע על מפתח הרישום שלו:

```
using System;
using System.Reflection;

public enum RegistryHives
{
   HKEY_CLASSES_ROOT,
   HKEY_CURRENT_USER,
   HKEY_LOCAL_MACHINE,
   HKEY_USERS,
   HKEY_CURRENT_CONFIG
}

public class RegistryKeyAttribute : Attribute
{
   public RegistryKeyAttribute(RegistryHives Hive, String ValueName)
   {
      this.Hive = Hive;
      this.ValueName = ValueName;
   }

   protected RegistryHives hive;
   public RegistryHives Hive
   {
      get { return hive; }
      set { hive = value; }
   }

   protected String valueName;
   public String ValueName
   {
      get { return valueName; }
      set { valueName = value; }
   }
}
```

```
class TestClass
{
   [RegistryKey(RegistryHives.HKEY_CURRENT_USER, "Foo")]
   public int Foo;

   public int Bar;
}

class FieldAttrApp
{
   public static void Main()
   {
      Type type = Type.GetType("TestClass");
      foreach(FieldInfo field in type.GetFields())
      {
         foreach (Attribute attr in field.GetCustomAttributes())
         {
            RegistryKeyAttribute registryKeyAttr =
                    attr as RegistryKeyAttribute;
            if (null != registryKeyAttr)
            {
               Console.WriteLine
                  ("{0} will be saved in {1}\\\\{2}",
                   field.Name,
                   registryKeyAttr.Hive,
                   registryKeyAttr.ValueName);
            }
         }
      }
   }
}
```

לא אסקור את כל הקוד, כי חלק ממנו מועתק מהדוגמה הקודמת. עם זאת, יש כאן מספר פרטים חשובים שראוי לדון בהם. תחילה, נשים לב שכפי שאובייקט MethodInfo מוגדר לקבלת מידע על שיטה מאובייקט Type, אובייקט FieldInfo מספק את אותה פונקציונליות לקבלת מידע על שדה מאובייקט Type. כמו בדוגמה הקודמת, אנו מתחילים בקבלת אובייקט Type המתייחס למחלקה TestClass. לאחר מכן אנו מבצעים איטרציה על המערך FieldInfo, ועבור כל אובייקט FieldInfo אנו מבצעים איטרציה על כל התכונות שלו עד שנמצא את התכונה הרצויה: registryKeyAttribute. אם וכאשר אנו מאתרים אותה, מדפיסים את שם השדה ומאחזרים מהתכונה את השדות Hive ו-ValueName.

# פרמטרים של תכונות

בדוגמאות הקודמות הסברתי איך אפשר לצרף תכונות באמצעות הבנאים שלהן. כעת אסקור מספר סוגיות העוסקים בבנאים של תכונות שטרם עסקתי בהם עד כה.

## פרמטרים לפי מיקום ופרמטרים לפי שם

בדוגמה FieldAttrApp בסעיף קודם ראינו תכונה בשם RegistryKeyAttribute. הבנאי שלה יכול להיראות כך:

```
public RegistryKeyAttribute(RegistryHives Hive, String ValueName)
```

ניתן לצרף את התכונה לשדה באופן דומה לזה:

```
[RegistryKey(RegistryHives.HKEY_CURRENT_USER, "Foo")]
public int Foo;
```

עד כאן הכל ברור. לבנאי יש שני פרמטרים, ובשניהם השתמשנו כאשר צירפנו תכונה זו לשדה. עם זאת, ניתן להקל כאן על המשתמש. אם רוב הזמן הפרמטר אינו עומד להשתנות, מדוע משתמש המחלקה צריך להקליד אותו בכל פעם? אפשר להגדיר ערכי ברירת מחדל באמצעות פרמטרים לפי מיקום, ופרמטרים לפי שם.

**פרמטרים לפי מיקום** (positional parameters) הם אלה שחובה לציין עבור בנאי התכונה בכל פעם שמשתמשים בתכונה. בדוגמת הקודמת, Hive ו-ValueName הם פרמטרים לפי מיקום. **פרמטרים לפי שם** (named parameters) אינם מוגדרים בבנאי של התכונה. אלה הם שדות ומאפיינים לא סטטיים. לפיכך, פרמטרים לפי שם מאפשרים להגדיר שדות ומאפיינים עבור התכונה בעת יצירת מופע, ללא צורך ליצור בנאי עבור כל צירוף אפשרי של שדות ומאפיינים שהלקוח רוצה להגדיר.

כל בנאי ציבורי יכול להגדיר רצף של פרמטרים לפי מיקום, כפי שאפשר עבור כל סוג של מחלקה. עם זאת, במקרה של תכונות, מרגע שנקבעו הפרמטרים לפי מיקום, המשתמש יכול לאחר מכן להתייחס לשדות או למאפיינים מסוימים עם התחביר FieldOrPropertyName=Value. כדי להמחיש זאת, הבה נשנה את התכונה RegistryKeyAttribute. בדוגמה זו, נהפוך את RegistryKeyAttribute.ValueName לפרמטר לפי מיקום יחידי, ואת RegistryKeyAttribute.Hive נהפוך לפרמטר לפי שם שהוא רשות (אופציונלי). נשאלת השאלה "איך מגדירים פרמטר לפי שם?" מכאן שרק פרמטרים לפי מיקום שנכללים בהגדרת הבנאי (וחובה להגדירם) יכולים לסלק את הפרמטר מהגדרת הבנאי. לאחר מכן, המשתמש יכול לפנות לכל שדה שאינו readonly, static או const, או לכל מאפיין הכולל שיטת גישה set (או setter) שאינו סטטי, כאל פרמטר לפי שם. לפיכך, בכדי להגדיר את RegistryKeyAttribute.Hive כפרמטר לפי שם, נסיר אותו מהגדרת הבנאי כי הוא כבר קיים כמאפיין ציבורי לקריאה וכתיבה:

```
public RegistryKeyAttribute(String ValueName)
```

כעת, המשתמש יכול לצרף את התכונה באחת מהדרכים הבאות:

```
[RegistryKey("Foo")]
[RegistryKey("Foo", Hive = RegistryHives.HKEY_LOCAL_MACHINE)]
```

כך מוענקת הגמישות להיות עם ערך ברירת מחדל עבור שדה, ובו בזמן לאפשר למשתמש לדרוס ערך זה אם יש צורך בכך. עם זאת, מתעוררת כאן בעיה. אם המשתמש אינו מגדיר את הערך של RegistryKeyAttribute.Hive, איך נקבע לו ערך ברירת מחדל? ובכן, אפשר אולי לבדוק אם הוא הוגדר בבנאי. אולם, הבעיה היא ש- RegistryKeyAttribute.Hive הוא enum (מנייה) הבנוי על סוג int, כלומר סוג ערך (value type). לפיכך, על פי ההגדרה, המהדר אתחל אותו עם הערך 0! אם נבחן את הערך של RegistryKeyAttribute.Hive בבנאי ונמצא שהוא שווה ל-0, לא נדע אם הערך הזה הוצב שם על ידי הקוד הקורא באמצעות פרמטר לפי שם, או שהמהדר אתחל אותו מכיון שהוא מסוג ערך. לרוע המזל, הדרך היחידה לעקוף בעיה זו היא לשנות את הקוד, כך שהערך 0 לא יתקבל כחוקי. ניתן לעשות זאת על ידי שינוי enum RegistryHives, כמוצג להלן:

```
public enum RegistryHives
{
   HKEY_CLASSES_ROOT = 1,
   HKEY_CURRENT_USER,
   HKEY_LOCAL_MACHINE,
   HKEY_USERS,
   HKEY_CURRENT_CONFIG
}
```

לאחר שינוי זה אנו יודעים בוודאות שהמקרה היחיד שבו RegistryKeyAttribute.Hive יכול להיות 0 הוא אם המהדר אתחל אותו ל-0 והמשתמש לא דרס ערך זה באמצעות פרמטר לפי שם. כעת נוכל לאתחל אותו עם קוד דומה לזה:

```
public RegistryKeyAttribute(String ValueName)
   {
      if (this.Hive == 0)
         this.Hive = RegistryHives.HKEY_CURRENT_USER;

      this.ValueName = ValueName;
   }
```

## שגיאות נפוצות עם פרמטרים לפי שם

כאשר רוצים להעביר הן פרמטרים לפי שם והן פרמטרים לפי מיקום, חייבים לציין תחילה את הפרמטרים לפי מיקום. לאחר מכן, כותבים את הפרמטרים לפי שם בכל סדר שהוא כאשר שם השדה או המאפיין מופיע לפניהם. לדוגמה, הקוד הבא גורם לשגיאת הידור:

```
//      This is an error because positional parameters can't follow
//      named parameters.
[RegistryKey(Hive=RegistryHives.HKEY_LOCAL_MACHINE, "Foo")]
```

בנוסף לכך, לא ניתן לציין שם עבור פרמטר לפי מיקום. כאשר המהדר מתרגם שימוש בתכונה, הוא ינסה תחילה לפרש את הפרמטרים לפי מיקום, ולאחר מכן ינסה לפרש את מה שנשאר – הפרמטרים לפי שם. הקוד הבא יגרום לשגיאת הידור, כי המהדר יכול לפרש כל פרמטר לפי שם. עם זאת, כאשר הוא מסיים לקרוא את הפרמטרים לפי שם, הוא אינו יכול למצוא פרמטרים לפי מיקום ומודיע: "No overload for method 'RegistryKeyAttribute' takes '0' arguments":

```
[RegistryKey(ValueName="Foo", Hive=RegistryHives.HKEY_LOCAL_MACHINE)]
```

לבסוף, פרמטרים לפי שם יכולים להיות כל שדה או מאפיין (כולל שיטה כותבת, setter) שיש אליהם גישה ציבורית ושאינם static או const.

## סוגים חוקיים של פרמטרים עבור תכונות

סוגי הפרמטרים לפי מיקום והפרמטרים לפי שם עבור מחלקה של תכונה מוגבלים לסוגים המפורטים להלן:

- bool, byte, char, double, float, int, long, short, string
- System.Type
- object
- סוג enum (מנייה), בתנאי שהוא, וכל סוג שבתוכו הוא מקונן, ניתנים לגישה ציבורית - כמו בדוגמה עם המנייה RegistryHives.
- מערך חד-מימדי מאחד הסוגים המפורטים לעיל.

סוגי הפרמטרים מוגבלים לאלה המופיעים ברשימה, ולכן לא ניתן להעביר מבני נתונים כמו מחלקות אל בנאי של תכונה. מגבלה זו הגיונית, כי תכונות מצורפות בזמן התכנון, ובנקודת זמן זו לא נוצר מופע של המחלקה (אובייקט). אפשר להשתמש בסוגים החוקיים המפורטים לעיל, כי הערכים שלהם ניתנים לקידוד קשיח בזמן התכנון.

# התכונה AttributeUsage

בנוסף לתכונות המותאמות לצירוף מידע נלווה לסוגי #C, ניתן להשתמש בתכונה AttributeUsage בכדי להגדיר איך רוצים שישתמשו בתכונות אלו. להלן מוסכמות הקריאה המתועדות של התכונה AttributeUsage:

```
[AttributeUsage(
validon,
AllowMultiple = allowmultiple,
Inherited = inherited
)]
```

כפי שרואים, קל להבחין בין הפרמטרים לפי מיקום לבין הפרמטרים לפי שם. אני ממליץ לך מאוד לתעד את התכונות על פי נוהג זה, כך שהמשתמש בתכונה שלך לא יצטרך לעבור על קוד המקור של מחלקת התכונה בכדי למצוא את השדות לקריאה/כתיבה ואת המאפיינים הציבוריים שניתן להשתמש בהם כתכונות לפי שם.

## הגדרת יעד התכונה

בתכונה AttributeUsage שהוצגה בסעיף קודם, שים לב שהפרמטר validon הוא פרמטר לפי מיקום (ולכן חובה לציין אותו). פרמטר זה מאפשר להגדיר את הסוגים שניתן לצרף אליהם את התכונה. למעשה, הפרמטר validon בתכונה AttributeUsage הוא מסוג AttributeTargets, שהוא מנייה המוגדרת כך:

```
public enum AttributeTargets
{
  Assembly      = 0x0001,
  Module        = 0x0002,
  Class         = 0x0004,
  Struct        = 0x0008,
  Enum          = 0x0010,
  Constructor   = 0x0020,
  Method        = 0x0040,
  Property      = 0x0080,
  Field         = 0x0100,
  Event         = 0x0200,
  Interface     = 0x0400,
  Parameter     = 0x0800,
  Delegate      = 0x1000,
  All = Assembly | Module | Class | Struct | Enum | Constructor |
      Method | Property | Field | Event | Interface | Parameter |
      Delegate,

  ClassMembers  =  Class | Struct | Enum | Constructor | Method |
      Property | Field | Event | Delegate | Interface,
}
```

שים לב שבשימוש עם AttributeTargets ניתן להגדיר AttributeTargets.All, כלומר לצרף את התכונה עם כל אחד מהסוגים המפורטים ב-AttributeTargets. זו גם ברירת המחדל אם לא מגדירים את התכונה AttributeUsage. לפיכך, אולי תתהה מדוע אי פעם תשתמש בערך validon. ובכן, ייתכן שתכונה זו מקבלת פרמטרים לפי שם, ואולי תרצה לשנות אחד מאלה. זכור שאם אתה משתמש בפרמטר לפי שם, עליך להציב לפניו את כל הפרמטרים לפי מיקום. הדבר מספק דרך נוחה לציין שאתה רוצה את שימוש ברירת המחדל של התכונה ב- AttributeTargets.All ועדיין מאפשר להגדיר את הפרמטרים לפי שם.

ובכן, מתי יש להגדיר את הפרמטר (AttributeTargets) validon ומדוע? משתמשים בו כאשר רוצים לשלוט באופן השימוש המדויק בתכונה. בדוגמאות הקודמות יצרנו את התכונה RemoteObjectAttribute שניתנת לשימוש עם מחלקות בלבד, את התכונה TransactionableAttribute עם שיטות בלבד ואת התכונה RegistryKeyAttribute שיש לה משמעות בשימוש עם שדות בלבד. אם רוצים להיות בטוחים שישתמשו בתכונות אלו לצירוף מידע רק לסוגים עבורם הן תוכננו, אפשר להגדירן באופן הבא (השמטנו שורות לשם קיצור ובהירות):

```
[AttributeUsage(AttributeTargets.Class)]
public class RemoteObjectAttribute : Attribute
{
   ...
}

[AttributeUsage(AttributeTargets.Method)]
public class TransactionableAttribute : Attribute
{
   ...
}

[AttributeUsage(AttributeTargets.Field)]
public class RegistryKeyAttribute : Attribute
{
   ...
}
```

נושא נוסף הקשור במנייה של AttributeTargets: ניתן לשלב חברים על ידי שימוש באופרטור "|". אם תכונה מסוימת ניתנת לצירוף גם לשדות וגם למאפיינים, עליך לצרף אליה את התכונה AttributeUsage, כך:

```
[AttributeUsage(AttributeTargets.Field | AttributeTargets.Property)]
```

## תכונות לשימוש יחיד ולשימוש מרובה

אפשר להשתמש ב-AttributeUsage בכדי להגדיר תכונות לשימוש יחיד או לשימוש מרובה. החלטה זו נוגעת למספר הפעמים שניתן להשתמש בתכונה אחת עם שדה יחיד. כברירת מחדל, כל התכונות הן לשימוש יחיד, ולכן הקוד הבא יגרום שגיאת הידור:

```
using System;
using System.Reflection;

public class SingleUseAttribute : Attribute
{
   public SingleUseAttribute(String str)
   {
   }
}

// ERROR: This results in a "duplicate attribute" compiler error.
[SingleUse("abc")]
[SingleUse("def")]
class MyClass
{
}

class SingleUseApp
{
   public static void Main()
   {
   }
}
```

בכדי לתקן בעיה זו, יש להגדיר בשורת AttributeUsage שרוצים לאפשר צירוף התכונה לסוג נתון באופן רב-פעמי. הקוד הבא אכן פועל:

```
using System;
using System.Reflection;

[AttributeUsage(AttributeTargets.All, AllowMultiple=true)]
public class SingleUseAttribute : Attribute
{
   public SingleUseAttribute(String str)
   {
   }
}

[SingleUse("abc")]
[SingleUse("def")]
class MyClass
{
```

```
}

class MultiUseApp
{
   public static void Main()
   {
   }
}
```

דוגמה מעשית שבה ניתן לנצל גישה זו היא RegistryKeyAttribute הנידונה בראשית פרק זה. סביר שאפשר לשמור שדה במספר מקומות ברישום המערכת, ולכן מצרפים את התכונה AttributeUsage עם הפרמטר לפי שם AllowMultiple כמו בקוד שלעיל.

## הגדרת חוקי הורשה לתכונה

הפרמטר האחרון עבור התכונה AttributeUsageAttribute הוא הדגל inherited, הקובע אם התכונה יכולה לעבור בהורשה. ערך ברירת המחדל הוא false. עם זאת, אם הדגל inherited מוגדר true, הוא תלוי בערך של הדגל AllowMultiple. אם הדגל inherited מוגדר true והדגל AllowMultiple מוגדר false, התכונה תדרוס את התכונה שעברה בהורשה. עם זאת, אם הדגל inherited מוגדר true והדגל AllowMultiple מוגדר גם כן true, התכונה מצטברת על החבר.

# Attribute Identifiers

התבונן בשורת הקוד הבאה ונסה לקבוע אם התכונה מצורפת לערך המוחזר או לשיטה:

```
class MyClass
{
   [HRESULT]
   public long Foo();
}
```

אם יש לך מעט ניסיון ב-COM, ידוע לך ש-HRESULT הוא הערך המוחזר הסטנדרטי עבור כל השיטות שאינן addRef או Release. עם זאת, קל לראות שאם שם התכונה חל גם על הערך המוחזר וגם על שם השיטה, המהדר אינו מסוגל לדעת מהי כוונתך. הנה מספר תסריטים אחרים שבהם המהדר לא יוכל להבין את כוונתך על פי ההקשר:

- שיטה מול ערך מוחזר.
- אירוע מול שדה מול מאפיין.
- נציג (delegate) מול ערך מוחזר.
- מאפיין מול שיטת גישה מול ערך מוחזר של שיטה קוראת מול פרמטר ערך של שיטה כותבת.

בכל אחד ממקרים אלה המהדר מקבל החלטה על פי מה שנחשב "מקובל ביותר". בכדי לדרוס החלטה זו, משתמשים במזהי תכונה, אשר כולם מפורטים להלן:

- assembly
- module
- type
- method
- property
- event
- field
- param
- return

משתמשים במזהה תכונה על ידי כתיבת המזהה עם נקודתיים לפני שם התכונה. בדוגמה MyClass, אם אתה רוצה לוודא שהמהדר יוכל לקבוע שהתכונה HRESULT חלה על הערך המוחזר ולא על השיטה, עליך להגדיר זאת כך:

```
class MyClass
{
   [return:HRESULT]
   public long Foo();
}
```

# סיכום

תכונות שפת #C מספקות מנגנון לסימון סוגים וחברים בעת התכנון, כדי שניתן יהיה לאחזר (לקרוא) אותן אחר כך על ידי reflection (שיקוף). הדבר מאפשר ליצור רכיבים עצמאיים העומדים בפני עצמם, ללא צורך לחפש דרכים למלא קבצי משאבים וקבועים בסיביות הדרושות. היתרון הוא השגת רכיב נייד יותר שגם קל יותר לפתח (לתכנת) וגם קל יותר לתחזק.

# פרק 9 ממשקים - Interfaces

המפתח להבנת ממשקים טמון בהשוואתם למחלקות. מחלקות הן אובייקטים בעלי מאפיינים ושיטות הפועלות על מאפיינים אלה. למרות שהן מציגות תכונות התנהגותיות מסוימות (שיטות), מחלקות הן **עצמים** בניגוד ל**התנהגויות**, וכך נכנסים ממשקים לתמונה. ממשקים מאפשרים להגדיר תכונות התנהגותיות, או יכולות, ולהחיל התנהגויות אלו על מחלקות, ללא התחשבות בהיררכיית המחלקה. דוגמה לכך הוא יישום מבוזר שבו חלק מהיישויות הן serializable (ניתנות לרישום). אלו עשויות לכלול את המחלקות Customer, Supplier ו-Invoice. ייתכן שמספר מחלקות אחרות, כמו MaintenanceView ו-Document אינן מוגדרות כ-serializable. איך תוכל להגדיר כ-serializable רק את המחלקות שאתה בוחר? דרך מובנת מאליה היא ליצור מחלקת בסיס בשם כמו Serializable. עם זאת, גישה זו לוקה בחיסרון אחד גדול. מסלול הורשה יחיד אינו הפתרון מאחר ואנו לא רוצים לשתף את כל ההתנהגויות של המחלקה. #C אינה תומכת בהורשה מרובה כך שלא ניתן לגזור מחלקה נתונה באופן סלקטיבי ממספר מחלקות (פרק 5). הפתרון ניתן באמצעות ממשקים (Interfaces). הם מאפשרים להגדיר קבוצה של שיטות ומאפיינים בעלי קירבה סמנטית שמחלקות יכולות ליישם מבלי להתחשב בהיררכיית המחלקה.

ניתן להסתכל על ממשקים כעל חוזים בין שני קטעי קוד שונים לגמרי. כלומר, מרגע שהוגדר הממשק והוגדרה מחלקה כמיישמת ממשק זה, מובטח ללקוחות של המחלקה שהיא מיישמת את כל השיטות המוגדרות בממשק. בקרוב תוכל לראות זאת במספר דוגמאות.

בפרק זה אסביר מדוע ממשקים הם חלק חשוב של #C, ובכלל של תכנות מבוסס-רכיבים. לאחר מכן נלמד איך להצהיר על ממשקים ולממש אותם ביישום #C. לסיום, נעסוק בפרטים נוספים הנוגעים לשימוש בממשקים ולהתגברות על הבעיות הטבועות בהורשה מרובה ובהתנגשות שמות.

הערה

כאשר מגדירים ממשק ומגדירים מחלקה כך שתשתמש בממשק זה, נאמר על המחלקה שהיא **מיישמת את הממשק או יורשת ממנו**. תוכל להשתמש בשני ביטויים אלה, ואף תמצא אותם משמשים באופן מתחלף בטקסטים אחרים. אישית, אני מאמין ש"**מיישמת**" היא מונח מתאים יותר מבחינה סמנטית - ממשקים הם התנהגויות מוגדרות ומחלקות מוגדרות כמיישמות התנהגות זו, בניגוד להורשה ממחלקה אחרת - אך שני הביטויים נכונים.

# השימוש בממשק

בכדי להבין היכן ממשקים שימושיים, נתייחס תחילה לבעיית תכנות מסורתית בפיתוח Microsoft Windows שאינה משתמשת בממשק, אך קיימים בה שני קטעי קוד נבדלים שצריכים לתקשר בצורה כללית. תאר לעצמך שאתה ראש צוות הפיתוח של לוח הבקרה (Control Panel) בחברת Microsoft. עליך לספק אמצעי כללי עבור כל יישומון לקוח לשילובו בלוח הבקרה כך שהסמל שלו יופיע בחלון לוח הבקרה ומשתמש הקצה יוכל לבצע את התוכנית. זכור שפונקציונליות זו תוכננה לפני הצגת COM, לכן איך תוכל לספק אמצעי לשילוב כל יישום עתידי בלוח הבקרה? הפתרון שנולד היה חלק סטנדרטי מפיתוח Windows במשך שנים.

אתה, כמוביל צוות פיתוח של לוח הבקרה, מתכנן ומתעד את הפונקציה (או הפונקציות) שהלקוח צריך ליישם, ביחד עם מספר חוקים. במקרה של יישומוני לוח הבקרה, Microsoft החליטה שכדי לכתוב יישומון עבור לוח הבקרה עליך ליצור ספריית קישור דינמי (DLL) המיישמת ומייצאת פונקציה בשם CPIApplet, לצרף אליה את הסיומת cpl ולמקם אותה בספריה System32 של Windows. (ב- Windows ME או Windows 98 תהיה זו הספריה Windows\System32, ב- Windows 2000 תהיה זו הספריה WINNT\System32). כאשר לוח הבקרה נטען, הוא טוען את כל הקבצים מסוג DLL עם סיומת cpl הנמצאים בספריה System32 (באמצעות הפונקציה LoadLibrary) ואז משתמש בפונקציה GetProcAddress בכדי לטעון את הפונקציה CPIApplet, ועל ידי זה מוודא שמילאת אחר החוקים ואתה יכול לתקשר עם לוח הבקרה באופן תקין.

כפי שציינתי, זהו מודל תכנות סטנדרטי ב-Windows לטיפול במצבים שבהם רוצים שקוד קיים יתקשר עם קוד עתידי בלתי ידוע באופן כללי. עם זאת, אין זה פתרון אלגנטי במיוחד, ויש לו בוודאות חסרונות משלו. החיסרון הגדול ביותר של טכניקה זו הוא שהיא מאלצת את הלקוח - קוד לוח הבקרה במקרה זה - לכלול קוד רב הנדרש לאימות.

לדוגמה, לוח הבקרה אינו יכול להניח סתם כך שכל קובץ עם הסיומת cpl הנמצא בספריה הוא DLL של Windows. כמו כן, לוח הבקרה צריך לוודא ש-DLL זה מכיל את פונקציות התיקון ואלו אכן מבצעות את מה שהוגדר בתיעוד. בנקודה זו נכנסים ממשקים לתמונה. הם מאפשרים לך ליצור אותו סידור חוזי בין קטעי קוד נבדלים אך בדרך מוכוונת-אובייקטים וגמישה יותר. בנוסף על כך, היות וממשקים הם חלק משפת #C, המהדר מבטיח שכאשר מחלקה מוגדרת כמיישמת ממשק נתון, היא אכן תיישם את כל מה שהממשק מציע.

ממשק ב-#C הוא מושג של מחלקה ראשונה, המצהיר על סוג ייחוס (reference type) הכולל הצהרה של שיטות בלבד. הביטוי "מושג של מחלקה ראשונה" אומר שהתכונה הנידונה מובנית בשפה, כלומר חלק אינטגרלי ממנה. במילים אחרות, התכונה לא "הולבשה" לאחר שהשפה נוצרה. כעת נרד לעומקם של הפרטים – מהם ממשקים ואיך מצהירים עליהם.

למפתחים ב- ++C הקוראים שורות אלו, ממשק הוא בעיקרו מחלקה מופשטת עם הצהרות של שיטות וירטואליות טהורות בלבד, בנוסף לסוגי חברי מחלקה אחרים של #C, כגון מאפיינים, אירועים וסדרנים.

# הצהרה על ממשקים

ממשקים יכולים להכיל שיטות, מאפיינים, סדרנים ואירועים, ואף אחד מאלה אינו מיושם בממשק עצמו. ניעזר בדוגמה בכדי להבין איך משתמשים בממשקים. נניח שאתה מתכנן עבור חברתך עורך (editor) המארח פקדי Windows שונים, וכותב את קוד העורך והשגרות המאשרות את הפקדים שהמשתמש מציב על טופס העורך. יתר אנשי צוות הפיתוח כותבים את הפקדים שהטפסים יארחו. קרוב לוודאי שאתה צריך לספק סוג מסוים של אישור ברמת הטופס. בעיתוי המתאים - לדוגמה, כאשר המשתמש מורה מפורשות לטופס לאשר את כל הפקדים, או במהלך עיבוד של טופס - הטופס יכול לגשת לכל אחד מהפקדים המצורפים אליו באמצעות לולאה "ולבדוק אותו או את תוכנו", או בצורה הולמת יותר, להורות לפקד לבדוק את עצמו.

ממשקים מצטיינים בהענקת יכולות כמו זו הנדרשת כאן של אישור, או בדיקת חוקיות לפקדים. הנה דוגמה פשוטה של ממשק המכיל שיטה יחידה בשם Validate:

```
interface IValidate
{
   bool Validate();
}
```

כעת תוכל לתעד את העובדה שניתן לאשר את הפקד אם הוא מיישם את הממשק IValidate.

הבה נבחן מספר היבטים של קטע הקוד הקודם. תחילה, אין צורך להגדיר מציין גישה כמו public בהצהרת שיטה של ממשק. למעשה, הוספת מציין גישה בהצהרת השיטה גורמת לשגיאת הידור. זאת מכיון שכל השיטות הנכללות בממשקים הן ציבוריות על פי ההגדרה. (מפתחי ++C אולי גם ישימו לב לכך שאין צורך להצהיר מפורשות על שיטה כווירטואלית טהורה על ידי צירוף 0= להצהרת השיטה. זאת מאחר ועל פי ההגדרה כל הממשקים הם מחלקות מופשטות).

בנוסף לשיטות, ממשקים יכולים להגדיר גם מאפיינים, סדרנים ואירועים, כמוצג להלן:

```
interface IExampleInterface
{
   // Example property declaration.
   int testProperty { get; }

   // Example event declaration.
   event testEvent Changed;

   // Example indexer declaration.
   string this[int index] { get; set; }
}
```

# מימוש ממשקים

מאחר וממשק מגדיר חוזה, כל מחלקה שמממשת ממשק **חייבת להגדיר את כל אחד ואחד מהפריטים של אותו ממשק** אחרת הידור הקוד ייכשל. בדוגמת IValidate הקודמת, מחלקת לקוח נדרשת לממש רק את שיטות הממשק. בדוגמה הבאה יש מחלקת בסיס הנקראת FancyControl וממשק הנקרא IValidate. לאחר מכן אני גוזר מהמחלקה FancyControl מחלקה בשם MyControl המממשת את הממשק IValidate. שים לב לתחביר ולאופן שבו ניתן להמיר את האובייקט MyControl לממשק IValidate בכדי להתייחס לחברים שלו.

```
using System;

public class FancyControl
{
   protected string Data;
      public string data
      {
         get
```

```
        {
            return this.Data;
        }
        set
        {
            this.Data = value;
        }
    }
}

interface IValidate
{
    bool Validate();
}

class MyControl : FancyControl, IValidate
{
    public MyControl()
    {
        data = "my grid data";
    }

    public bool Validate()
    {
        Console.WriteLine("Validating...{0}", data);
                return true;
    }
}

class InterfaceApp
{
    public static void Main()
    {
        MyControl myControl = new MyControl();

        // Call a function to place the control on
        // the form. Now, when the editor needs to
        // validate the control, it can do
        // the following:

        IValidate val = (IValidate)myControl;
        bool success = val.Validate();
    Console.WriteLine("The validation of '{0}' was {1}successful",
                    myControl.data,
                    (true == success ? "" : "not "));
    }
}
```

באמצעות הגדרת המחלקה והממשק הקודמים, העורך יכול לקבל מידע מהפקד האם הוא מממש את הממשק IValidate. (תלמד איך לבצע שלב זה בסעיף הבא). אם הפקד מממש ממשק זה, העורך יכול לאשר ואז לקרוא לשיטות הממשק הממומשות. כאן יש אולי מניע לשאול "מדוע שלא נגדיר מחלקת בסיס לשימוש עם העורך, המכילה פונקציה וירטואלית בשם Validate, והעורך יקבל רק פקדים שנגזרו ממחלקת בסיס זו?"

פתרון זה הוא אמנם בר-ביצוע אך מגביל קשות. נניח שאתה יוצר פקדים משלך וכולם נגזרים מאותה מחלקת בסיס היפותטית. כתוצאה מכך, הם כולם מממשים את השיטה הווירטואלית Validate הזו. זה עובד עד שיום אחד אתה מגלה פקד מלהיב ואתה רוצה להשתמש בו עם העורך. נניח שמדובר בפקד רשת שנכתב על ידי מישהו אחר. פקד זה לכשעצמו לא ייגזר מאותה מחלקת הבסיס המסורתית עבור פקדים של העורך. ב- C++ הפתרון הוא להשתמש בהורשה מרובה ולגזור את מחלקת הרשת ממחלקת הרשת של צד ג' וממחלקת הבסיס של העורך גם יחד. עם זאת, C# אינה תומכת בהורשה מרובה.

בעזרת ממשקים ניתן לממש תכונות התנהגותיות מרובות במחלקה אחת. ב-C# ניתן לגזור ממחלקה אחת, ובנוסף לפונקציונליות שעוברת בהורשה ניתן לממש ממשקים ככל הנדרש למחלקה. לדוגמה, אם תרצה שיישום העורך יאשר את תכולת הפקד, קשר אותו עם מסד נתונים ורשום את תכולתו בדיסק. כך תצהיר על המחלקה שלך:

```
public class MyGrid : ThirdPartyGrid, IValidate,
                ISerializable, IDataBound
{

}
```

כפי שהבטחתי, הסעיף הבא מסביר איך קטע קוד יכול לדעת אם מחלקה מממשת ממשק נתון.

## אחזור מידע אודות יישום ממשקים באמצעות is

בדוגמה InterfaceApp ראית את הקוד הבא, ששימש להמרת אובייקט (MyControl) לאחד מהממשקים הממומשים שלו (IValidate), ולאחר מכן לקריאה לאחד מחברי ממשק זה (Validate):

```
MyControl myControl = new MyControl();
:
IValidate val = (IValidate)myControl;
bool success = val.Validate();
```

מה קורה אם לקוח מנסה להשתמש במחלקה כאילו היא מממשת שיטה שבעצם אינה ממומשת על ידה?

הידור הדוגמה הבאה יצליח מאחר ו-ISerializable הוא ממשק חוקי. בכל זאת, בזמן ריצה מתעורר החריג System.InvalidCastException מאחר ו-MyGrid אינו מממש את הממשק ISerializable. היישום יפסיק את ביצועו אלא אם החריג נתפס בצורה מפורשת.

```
using System;

public class FancyControl
{
   protected string Data;
   public string data
   {
      get
      {
         return this.Data;
      }
      set
      {
         this.Data = value;
      }
   }
}

interface ISerializable
{
   bool Save();
}

interface IValidate
{
   bool Validate();
}

class MyControl : FancyControl, IValidate
{
   public MyControl()
   {
      data = "my grid data";
   }

   public bool Validate()
   {
      Console.WriteLine("Validating...{0}", data);
      return true;
   }
}
```

```
class IsOperator1App
{
   public static void Main()
   {
      MyControl myControl = new MyControl();

      ISerializable ser = (ISerializable)myControl;

      // NOTE: This will throw a System.InvalidateCastException
      // because the class does not implement the ISerializable
      // interface.
      bool success = ser.Save();

      Console.WriteLine("The saving of '{0}' was {1}successful",
 myControl.data,
 (true == success ? "" : "not "));
   }
}
```

כמובן שלכידת החריג אינה משנה את העובדה שהקוד אותו אתה רוצה לבצע לא יתבצע במקרה זה. חייבת להיות דרך לקבל מידע מהאובייקט אודות הממשק **לפני** שמנסים להמיר אותו. דרך אחת לבצע זאת היא באמצעות האופרטור **is**. אופרטור זה מאפשר לבדוק בזמן ריצה אם סוג אחד תואם לסוג אחר. הוא מקבל את הצורה הבאה, כאשר expression הוא סוג ייחוס (reference type):

```
expression is type
```

התוצאה של האופרטור is הוא ערך בוליאני ולכן הוא יכול לשמש במשפטי תנאי. בדוגמה הבאה שיניתי את הקוד כך שתתבצע בדיקת התאמה בין המחלקה MyControl והממשק ISerializable לפני שמנסים להשתמש בשיטה ISerializable:

```
using System;

public class FancyControl
{
   protected string Data;
   public string data
   {
      get
      {
         return this.Data;
      }
      set
      {
         this.Data = value;
      }
   }
}
```

```
interface ISerializable
{
   bool Save();
}

interface IValidate
{
   bool Validate();
}

class MyControl : FancyControl, IValidate
{
   public MyControl()
   {
      data = "my grid data";
   }

   public bool Validate()
   {
      Console.WriteLine("Validating...{0}", data);
      return true;
   }
}

class IsOperator2App
{
   public static void Main()
   {
      MyControl myControl = new MyControl();

      if (myControl is ISerializable)
      {
         ISerializable ser = (ISerializable)myControl;
         bool success = ser.Save();

         Console.WriteLine("The saving of '{0}' was {1}successful",
         myControl.data,
         (true == success ? "" : "not "));
      }
      else
      {
       Console.WriteLine("The ISerializable interface is not implemented.");
      }
   }
}
```

כעת לאחר שראינו איך האופרטור is מאפשר לבדוק את תקפות ההתאמה בין שני סוגים בכדי להבטיח שימוש נכון, הבה נבחן את אחד מקרוביו - האופרטור as - ונשווה בין השניים.

## אחזור מידע אודות מימוש ממשקים באמצעות as

התבוננות מקרוב בקוד MSIL שנוצר מהדוגמה IsOperator2App הקודמת (מוצג לאחר פיסקה זו) חושפת בעיה אחת עם האופרטור is. ניתן לראות שקוד הפעולה isinst נקרא מייד לאחר הקצאת האובייקט והתקנת המחסנית. קוד הפעולה isinst נוצר על ידי המהדר עבור האופרטור is של #C. קוד פעולה זה בוחן אם האובייקט הוא מופע של מחלקה או ממשק. מספר שורות לאחר מכן - בהנחה שמבחן התנאי עבר - המהדר יצר את קוד הפעולה castclass. קוד IL זה מבצע אימות משלו באופן מעט שונה מקוד הפעולה isinst. כתוצאה מכך נבדקת התקפות של המרה זו פעמיים ולכן הקוד IL שנוצר מבצע עבודה לא יעילה.

```
.method public hidebysig static void Main() il managed
{
  .entrypoint
  // Code size       72 (0x48)
  .maxstack  4
  .locals (class MyControl V_0,
           class ISerializable V_1,
           bool V_2)
  IL_0000:  newobj     instance void MyControl::.ctor()
  IL_0005:  stloc.0
  IL_0006:  ldloc.0
  IL_0007:  isinst     ISerializable
  IL_000c:  brfalse.s  IL_003d
  IL_000e:  ldloc.0
  IL_000f:  castclass  ISerializable
  IL_0014:  stloc.1
  IL_0015:  ldloc.1
  IL_0016:  callvirt   instance bool ISerializable::Save()
  IL_001b:  stloc.2
  IL_001c:  ldstr      "The saving of '{0}' was {1}successful"
  IL_0021:  ldloc.0
  IL_0022:  call       instance class System.String FancyControl::get_data()
  IL_0027:  ldloc.2
  IL_0028:  brtrue.s   IL_0031
  IL_002a:  ldstr      "not "
  IL_002f:  br.s       IL_0036
  IL_0031:  ldstr      ""
  IL_0036:  call   void [mscorlib]System.Console::WriteLine(class System.String,
                                                            class System.Object,
                                                            class System.Object)
```

```
  IL_003b:  br.s   IL_0047
  IL_003d:  ldstr  "The ISerializable interface is not implemented."
  IL_0042:  call  void [mscorlib]System.Console::WriteLine(class System.String)
  IL_0047:  ret
} // end of method IsOperator2App::Main
```

נוכל לייעל תהליך אימות זה על ידי שימוש באופרטור **as**. אופרטור זה מבצע המרה בין שני סוגים תואמים ומקבל את הצורה הבאה, כאשר expression הוא סוג ייחוס (reference type) כלשהו:

```
object = expression as type
```

אפשר לראות את האופרטור as כצירוף של האופרטור is והמרה (אם שני הסוגים הנידונים הם תואמים). הבדל חשוב בין האופרטור as והאופרטור is הוא שאם expression ו-type אינם תואמים, האופרטור as קובע לאובייקט ערך null במקום להחזיר ערך בוליאני. כעת נוכל לשכתב את הדוגמה שלנו כך שתפעל בדרך יעילה יותר:

```
using System;

public class FancyControl
{
   protected string Data;
   public string data
   {
      get
      {
         return this.Data;
      }
      set
      {
         this.Data = value;
      }
   }
}

interface ISerializable
{
   bool Save();
}

interface IValidate
{
   bool Validate();
}
```

```
class MyControl : FancyControl, IValidate
{
   public MyControl()
   {
      data = "my grid data";
   }

   public bool Validate()
   {
      Console.WriteLine("Validating...{0}", data);
      return true;
   }
}

class AsOperatorApp
{
   public static void Main()
   {
      MyControl myControl = new MyControl();

      ISerializable ser = myControl as ISerializable;
      if (null != ser)
      {
         bool success = ser.Save();

         Console.WriteLine("The saving of '{0}' was {1}successful",
myControl.data,
(true == success ? "" : "not "));
      }
      else
      {
        Console.WriteLine("The ISerializable interface is not implemented.");
      }
   }
}
```

כעת האימות הנדרש בכדי להבטיח המרה חוקית מתבצע פעם אחת בלבד, אשר ללא ספק הוא יעיל הרבה יותר. בנקודה זו, נתבונן שוב בקוד MSIL לבחון את ההשפעה של השימוש באופרטור as:

```
.method public hidebysig static void Main() il managed
{
  .entrypoint
  // Code size       67 (0x43)
  .maxstack  4
```

```
  .locals (class MyControl V_0,
           class ISerializable V_1,
           bool V_2)
  IL_0000:  newobj     instance void MyControl::.ctor()
  IL_0005:  stloc.0
  IL_0006:  ldloc.0
  IL_0007:  isinst     ISerializable
  IL_000c:  stloc.1
  IL_000d:  ldloc.1
  IL_000e:  brfalse.s  IL_0038
  IL_0010:  ldloc.1
  IL_0011:  callvirt   instance bool ISerializable::Save()
  IL_0016:  stloc.2
  IL_0017:  ldstr      "The saving of '{0}' was {1}successful"
  IL_001c:  ldloc.0
  IL_001d:  call       instance class System.String FancyControl::get_data()
  IL_0022:  ldloc.2
  IL_0023:  brtrue.s   IL_002c
  IL_0025:  ldstr      "not "
  IL_002a:  br.s       IL_0031
  IL_002c:  ldstr      ""
  IL_0031:  call   void [mscorlib]System.Console::WriteLine(class System.String,
                                                            class System.Object,
                                                            class System.Object)
  IL_0036:  br.s       IL_0042
  IL_0038:  ldstr      "The ISerializable interface is not implemented."
  IL_003d:  call   void [mscorlib]System.Console::WriteLine(class System.String)
  IL_0042:  ret
} // end of method AsOperatorApp::Main
```

# שיוך מפורש של שם חבר ממשק

עד כה ראינו מחלקות המממשות ממשקים על ידי הגדרת מציין הגישה public ולאחריו חתימת השיטה של הממשק. עם זאת, לעיתים רוצים (או אפילו חייבים) לשייך באופן מפורש את שמו של החבר לשם הממשק. בחלק זה נבחן שתי סיבות שכיחות לעשות כך.

## הסתרת שמות באמצעות ממשקים

הדרך השכיחה ביותר לקרוא לשיטה ממומשת מתוך ממשק היא להמיר מופע של מחלקה זו לסוג הממשק ואז לקרוא לשיטה הרצויה. בעוד שטכניקה זו היא חוקית ורבים (כולל אני) משתמשים בה, מבחינה טכנית אין צורך להמיר את האובייקט לממשק הממומש שלו בכדי לקרוא לשיטות של ממשק זה. זאת מכיון שאם מחלקה

מממשת שיטות של ממשק, שיטות אלו הן גם שיטות ציבוריות של המחלקה. התבונן בקוד #C הבא ובמיוחד בשיטה main בכדי להבין זאת:

```
using System;

public interface IDataBound
{
   void Bind();
}

public class EditBox : IDataBound
{
   // IDataBound implementation.
   public void Bind()
   {
      Console.WriteLine("Binding to data store...");
   }
}

class NameHiding1App
{
   // Main entry point.
   public static void Main()
   {
      Console.WriteLine();

      EditBox edit = new EditBox();
      Console.WriteLine("Calling EditBox.Bind()...");
      edit.Bind();

      Console.WriteLine();

      IDataBound bound = (IDataBound)edit;
      Console.WriteLine("Calling (IDataBound)EditBox.Bind()...");
      bound.Bind();
   }
}
```

דוגמה זו מפיקה כעת את הפלט הבא:

```
Calling EditBox.Bind()...
Binding to data store...

Calling (IDataBound))EditBox.Bind()...
Binding to data store...
```

שים לב לכך, שלמרות שהיישום קורא לשיטה הממומשת Bind בשתי דרכים שונות - אחת עם המרה ואחת ללא המרה - עיבוד השיטה Bind בשתי הקריאות מתבצע בצורה נכונה. ממבט ראשון נראה כי היכולת לקרוא ישירות לשיטה הממומשת ללא המרת האובייקט לממשק היא דבר טוב, אולם ישנם זמנים שבהם זה פחות מרצוי. הסיבה הבולטת ביותר היא מימושם של מספר ממשקים, כאשר כל אחד מהם עשוי להכיל מספר רב של חברים. במקרים כאלה עלולים "לזהם" במהירות את מרחב השמות הציבורי של המחלקה עם חברים שאין להם כל משמעות מחוץ לטווח ההכרה של המחלקה הממומשת. ניתן למנוע מחברי ממשק ממומשים מלהיות חברי מחלקה ציבוריים באמצעות טכניקה הנקראת **הסתרת שמות** (name hiding).

במשמעותה הפשוטה, הסתרת שמות היא היכולת להסתיר את שמו של חבר שעבר בהורשה מכל קוד שנמצא מחוץ למחלקה הנגזרת או הממומשת (מה שמקובל להתייחס אליו כאל **העולם החיצון**). נתייחס לדוגמה הקודמת שבה המחלקה EditBox צריכה לממש את הממשק IDataBound, והפעם נניח שהמחלקה EditBox אינה רוצה לחשוף את שיטות IDataBound לעולם החיצון. במקום זאת, היא זקוקה לממשק זה לשימושה העצמי, או שהמתכנת פשוט אינו רוצה "לבלגן" את מרחב השמות של המחלקה עם מספר גדול של שיטות שלקוח טיפוסי לא ישתמש בהם. בכדי להסתיר חבר ממשק ממומש, פשוט צריך להסיר את המציין public של החבר ולשייך את שם החבר לשם הממשק, הנה כך:

```
using System;

public interface IDataBound
{
   void Bind();
}

public class EditBox : IDataBound
{
   // IDataBound implementation.
   void IDataBound.Bind()
   {
      Console.WriteLine("Binding to data store...");
   }
}

class NameHiding2App
{
   public static void Main()
   {
      Console.WriteLine();
```

```
    EditBox edit = new EditBox();
    Console.WriteLine("Calling EditBox.Bind()...");

    // ERROR: This line won't compile because
    // the Bind method no longer exists in the
    // EditBox class's namespace.
    edit.Bind();

    Console.WriteLine();

    IDataBound bound = (IDataBound)edit;
    Console.WriteLine("Calling (IDataBound)EditBox.Bind()...");

    // This is OK because the object was cast to
    // IDataBound first.
    bound.Bind();
}
}
```

הידור הקוד הקודם לא יצליח היות ושם החבר Bind אינו יותר חלק של המחלקה EditBox. לפיכך, טכניקה זו מאפשרת להסיר חבר ממרחב השמות של המחלקה ועדיין לאפשר גישה מפורשת באמצעות פעולת המרה.

נקודה שיש לזכור היא שכאשר מסתירים שם חבר, לא ניתן להשתמש במציין גישה. אם תנסה להשתמש במציין גישה עם שם חבר ממומש במחלקה תקבל שגיאת זמן-הידור. אולי זה נראה מוזר, אבל קח בחשבון שעצם העניין בהסתרת חבר הוא למנוע ממנו להיות גלוי מחוץ למחלקה הנוכחית. היות ומצייני גישה קיימים רק בכדי להגדיר את רמת הזמינות מחוץ למחלקת הבסיס, הרי שאין להם כל משמעות כאשר משתמשים בהסתרת שמות.

## מניעת אי-בהירות עם שמות

אחת הסיבות העיקריות לכך ש-C# אינה תומכת בהורשה מרובה היא הבעיה של התנגשות שמות, הנגרמת מאי-בהירות של שמות. למרות ש-C# אינה תומכת בהורשה מרובה ברמת האובייקט (גזירה ממחלקה), היא כן תומכת בהורשה ממחלקה אחת וביישום הנוסף של ממשקים מרובים. עם זאת, יש לעוצמה זו גם מחיר: התנגשות שמות.

בדוגמה הבאה ישנם שני ממשקים, ISerializable ו-IDataStroe, אשר תומכים בקריאה ואחסון נתונים בשני פורמטים שונים – האחד כאובייקט לדיסק בצורה בינארית והשני למסד נתונים. הבעיה היא ששניהם מכילים שיטות הנקראות SaveData:

```
using System;

interface ISerializable
{
   void SaveData();
}

interface IDataStore
{
   void SaveData();
}

class Test : ISerializable, IDataStore
{
   public void SaveData()
   {
      Console.WriteLine("Test.SaveData called");
   }
}

class NameCollisions1App
{
   public static void Main()
   {
      Test test = new Test();

      Console.WriteLine("Calling Test.SaveData()");
      test.SaveData();
   }
}
```

בזמן כתיבת הקוד הוא אכן עובר הידור מוצלח. עם זאת, נאמר לי כי בגירסה עתידית של מהדר #C, קוד זה יגרום לשגיאת זמן-הידור בגלל אי-הבהירות הקיימת בשיטה המיושמת SaveData. ללא קשר לכך אם הקוד מהודר בהצלחה, בזמן ריצה יש בעיה מאחר וההתנהגות הנגרמת מהקריאה לשיטה SaveData אינה ברורה למתכנת המנסה להשתמש במחלקה זו. האם מקבלים את SaveData הרושמת את האובייקט לדיסק או את SaveData השומרת במסד הנתונים?

בנוסף לכך, התבונן בקוד הבא:

```
using System;

interface ISerializable
{
   void SaveData();
}
```

```
interface IDataStore
{
   void SaveData();
}

class Test : ISerializable, IDataStore
{
   public void SaveData()
   {
      Console.WriteLine("Test.SaveData called");
   }
}

class NameCollisions2App
{
   public static void Main()
   {
      Test test = new Test();

      if (test is ISerializable)
      {
         Console.WriteLine("ISerializable is implemented");
      }

      if (test is IDataStore)
      {
         Console.WriteLine("IDataStore is implemented");
      }
   }
}
```

כאן המבחן באמצעות האופרטור is מצליח עבור שני הממשקים, מה שאומר ששניהם ממומשים למרות שידוע לנו שאין זה המקרה! במקרה זה אפילו המהדר פולט את האזהרות הבאות במהלך ההידור של דוגמה זו:

```
NameCollisions2.cs(27,7): warning CS0183: The given expression is
always of the provided ('ISerializable') type
NameCollisions2.cs(32,7): warning CS0183: The given expression is
always of the provided ('IDataStore') type
```

הבעיה היא שהמחלקה מימשה גירסה הרושמת את האובייקט או גירסה של מסד הנתונים (אך לא את שתיהן). עם זאת, אם הלקוח בודק את מימושו של אחד מהממשקים - הבדיקה תצליח עבור שניהם יחד - ומנסה להשתמש באותו ממשק שלא מומש באמת, הדבר יגרום לתוצאות לא צפויות.

ניתן לעקוף בעיה זו באמצעות שיוך מפורש של שם חבר ממשק: מסירים את מציין הגישה, ומוסיפים לפני שם החבר - במקרה זה, SaveData - את שם הממשק:

```
using System;

interface ISerializable
{
   void SaveData();
}

interface IDataStore
{
   void SaveData();
}

class Test : ISerializable, IDataStore
{
   void ISerializable.SaveData()
   {
      Console.WriteLine("Test.ISerializable.SaveData called");
   }
   void IDataStore.SaveData()
   {
      Console.WriteLine("Test.IDataStore.SaveData called");
   }
}

class NameCollisions3App
{
   public static void Main()
   {
      Test test = new Test();

      if (test is ISerializable)
      {
         Console.WriteLine("ISerializable is implemented");
         ((ISerializable)test).SaveData();
      }

      Console.WriteLine();

      if (test is IDataStore)
      {
         Console.WriteLine("IDataStore is implemented");
         ((IDataStore)test).SaveData();
      }
   }
}
```

כעת אין אי-בהירות באשר לאיזו שיטה תקרא. שתי השיטות ממומשות עם שמותיהן המשויכים במלואם, והפלט המתקבל בביצוע היישום הוא כמצופה:

```
ISerializable is implemented
Test.ISerializable.SaveData called

IDataStore is implemented
Test.IDataStore.SaveData called
```

# ממשקים והורשה

קיימות שתי בעיות שכיחות הקשורות לממשקים והורשה. הבעיה הראשונה, המוצגת בדוגמת הקוד שלהלן, עוסקת בסוגיה של גזירה ממחלקת בסיס המכילה שם שיטה הזהה לשם של שיטת ממשק שהמחלקה צריכה לממש.

```
using System;

public class Control
{
   public void Serialize()
   {
      Console.WriteLine("Control.Serialize called");
   }
}

public interface IDataBound
{
   void Serialize();
}

public class EditBox : Control, IDataBound
{
}

class InterfaceInh1App
{
   public static void Main()
   {
      EditBox edit = new EditBox();
      edit.Serialize();
   }
}
```

כידוע, בכדי לממש ממשק חייבים לספק הגדרה עבור כל חבר הכלול בהצהרה של אותו ממשק. עם זאת, בדוגמה הקודמת אין אנו עושים כך ובכל זאת הקוד מהודר בהצלחה! הסיבה לכך היא שמהדר #C מחפש אם השיטה Serialize מיושמת במחלקה EditBox ומוצא אותה. אולם, המהדר שוגה בקביעה שזו השיטה הממומשת. השיטה Serialize שאותה מוצא המהדר היא השיטה Serialize שעברה בהורשה מהמחלקה Control ואינה מימוש ממשי של השיטה IDataBound.Serialize. לכן למרות שהידור הקוד מצליח, הוא לא יפעל כמצופה, כפי שנראה בהמשך.

כעת נעשה את הדברים מעניינים קצת יותר. שים לב שהקוד הבא תחילה בודק - באמצעות האופרטור as - שהממשק ממומש ואז מנסה לקרוא לשיטה הממומשת Serialize. הידור הקוד מצליח ופועל. עם זאת, כפי שאנו יודעים, המחלקה EditBox אינה מממשת באמת את השיטה Serialize כתוצאה מההורשה של IDataBound. ל-EditBox יש כבר את השיטה Serialize (שעברה בהורשה) מהמחלקה Control. פירושו של דבר, שהלקוח ככל הנראה לא יקבל את התוצאות המצופות.

```
using System;

public class Control
{
   public void Serialize()
   {
      Console.WriteLine("Control.Serialize called");
   }
}

public interface IDataBound
{
   void Serialize();
}

public class EditBox : Control, IDataBound
{
}

class InterfaceInh2App
{
   public static void Main()
   {
      EditBox edit = new EditBox();

   IDataBound bound = edit as IDataBound;
   if (bound != null)
   {
      Console.WriteLine("IDataBound is supported...");
      bound.Serialize();
   }
```

```
    else
    {
        Console.WriteLine("IDataBound is NOT supported...");
    }
    }
}
```

בעיה פוטנציאלית אחרת שיש להיזהר ממנה היא כאשר למחלקה נגזרת יש שיטה עם אותו שם כמו של שיטת ממשק הממומשת במחלקת הבסיס. כך זה נראה בקוד:

```
using System;

interface ITest
{
    void Foo();
}

// Base implements ITest.
class Base : ITest
{
    public void Foo()
    {
        Console.WriteLine("Base.Foo (ITest implementation)");
    }
}

class MyDerived : Base
{
    public new void Foo()
    {
        Console.WriteLine("MyDerived.Foo");
    }
}

public class InterfaceInh3App
{
    public static void Main()
    {
        MyDerived myDerived = new MyDerived();
        myDerived.Foo();

        ITest test = (ITest)myDerived;
        test.Foo();
    }
}
```

קוד זה מפיק את הפלט הבא:

```
MyDerived.Foo
Base.Foo (ITest implementation)
```

במצב זה, המחלקה Base מממשת את הממשק ITest ואת השיטה Foo שלו. עם זאת, המחלקה MyDerived נגזרת מהמחלקה Base כמחלקה חדשה וממשת שיטת Foo חדשה עבור מחלקה זו. איזו שיטת Foo נקראת? זה תלוי מה הייחוס שיש לך. אם יש ייחוס לאובייקט MyDerived, השיטה Foo שלו תיקרא. זאת מכיון שלמרות שלאובייקט MyDerived יש מימוש של ITest.Foo שעבר בהורשה, זמן הריצה יבצע את השיטה MyDerived.Foo כי מילת המפתח new מציינת דריסה של השיטה שעברה בהורשה.

עם זאת, כאשר ממירים מפורשות את אובייקט MyDerived לממשק ITest, המהדר מתרגם זאת למימוש של הממשק. המחלקה MyDerived מכילה שיטה בעלת אותו שם אך זה לא מה שהמהדר מחפש. כאשר ממירים אובייקט לממשק, המהדר עובר לאורך עץ ההורשה עד שנמצאת מחלקה המכילה את הממשק ברשימת הבסיס שלה. זו הסיבה מדוע שתי שורות הקוד האחרונות בשיטה main גורמות לקריאה לשיטה הממומשת Foo של ITest.

מלכודות פוטנציאליות אלו הקשורות להתנגשות שמות והורשת ממשקים רק חיזקו את המלצתי התקיפה: תמיד המר את האובייקט לממשק שאתה מנסה להשתמש בחבר שלו.

# איחוד ממשקים

תכונה רבת עוצמה אחרת של #C היא היכולת לאחד שניים או יותר ממשקים כך שהמחלקה צריכה רק ליישם את התוצאה המאוחדת. לדוגמה, נניח שרוצים ליצור מחלקת TreeView חדשה המממשת את הממשקים IDragDrop ו-ISerializable גם יחד. מאחר וסביר להניח שפקדים אחרים, כגון ListView ו-ListBox, ירצו גם כן לאחד רכיבים אלו, אולי נרצה לאחד את הממשקים IDragDrop ו-ISerializable לממשק יחיד:

```
using System;

public class Control
{
}

public interface IDragDrop
{
   void Drag();
   void Drop();
}
```

```
public interface ISerializable
{
   void Serialize();
}

public interface ICombo : IDragDrop, ISerializable
{
   // This interface doesn't add anything new in
   // terms of behavior as its only purpose is
   // to combine the IDragDrop and ISerializable
   // interfaces into one interface.
}

public class MyTreeView : Control, ICombo
{
   public void Drag()
   {
      Console.WriteLine("MyTreeView.Drag called");
   }

   public void Drop()
   {
      Console.WriteLine("MyTreeView.Drop called");
   }

   public void Serialize()
   {
      Console.WriteLine("MyTreeView.Serialize called");
   }
}

class CombiningApp
{
   public static void Main()
   {
      MyTreeView tree = new MyTreeView();
      tree.Drag();
      tree.Drop();
      tree.Serialize();
   }
```

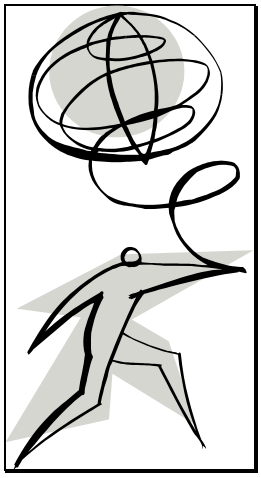
בעזרת היכולת לאחד ממשקים, ניתן לא רק לפשט את היכולת לקבץ ממשקים בעלי קירבה סמנטית לממשק יחיד, אלא גם להוסיף שיטות נוספות לממשק "המורכב" החדש במידת הצורך.

# סיכום

ממשקים ב-#C מאפשרים לפתח מחלקות המסוגלות להיות שותפות בתכונות מבלי להיות חלק מאותה היררכיית מחלקה. לממשקים יש תפקיד מיוחד בפיתוח בשפת #C, מאחר ו-#C אינה תומכת בהורשה מרובה. כדי להשתתף בשיטות ובמאפיינים בעלי קירבה סמנטית, מחלקות יכולות לממש ממשקים מרובים. בנוסף לכך, ניתן להשתמש באופרטורים is ו-as בכדי לקבוע אם ממשק מסוים ממומש על ידי אובייקט, דבר שיכול למנוע שגיאות הקשורות לשימוש בחברים של ממשקים. לבסוף, ניתן להשתמש בכינוי מפורש של חברי ממשק ובהסתרת שמות בכדי לשלוט במימושם של ממשקים ולעזור במניעת שגיאות.

## חלק 3

# כתיבת קוד



# פרק 10 ביטויים ואופרטורים

בפרק זה נעסוק בחלק הבסיסי ביותר של כל שפת תכנות: יכולתה לבטא הצבות והשוואות באמצעות אופרטורים. נלמד מהם אופרטורים ואיך נקבעת קדימות האופרטור ב-#C. בהמשך נעסוק ביתר פירוט בקטגוריות מסוימות של ביטויים לביצוע דברים כמו פעולות מתמטיות, הצבת ערכים והשוואה בין אופרנדים.

## הגדרת אופרטורים

אופרטור הוא סימן המציין פעולה שיש לבצע על אחד או יותר ארגומנטים. פעולה זו מייצרת תוצאה. התחביר שבו משתמשים עם אופרטורים הוא מעט שונה מזה של קריאה לשיטות, אבל פורמט #C של ביטוי המשתמש באופרטורים צריך להוות עבורך טבע שני. כמו אופרטורים ברוב שפות התכנות האחרות, הסמנטיקה של אופרטור #C נוהגת על פי החוקים והסימנים הבסיסיים שלמדנו כילדים בשיעורי חשבון. האופרטורים הבסיסיים ביותר של #C כוללים כפל (*), חילוק (/), חיבור ופלוס אונארי (+), חיסור ומינוס אונארי (-), שארית (%) והצבה (=).

אופרטורים תוכננו במיוחד ליצירת ערך חדש מהערך שעליו פועלים. הערכים שעליהם פועלים נקראים **אופרנדים** (operands). תוצאת הפעולה חייבת להיות מאוחסנת במקום כלשהו בזיכרון. במספר מקרים, הערכים שנוצרים על ידי פעולה מאוחסנים במשתנה המכיל את אחד מהאופרנדים המקוריים. מהדר #C מייצר הודעת שגיאה אם משתמשים באופרטור ולא ניתן לקבוע או לאחסן ערך חדש. בדוגמה הבאה, הקוד אינו גורם לשום שינוי. המהדר יוצר הודעת שגיאה מאחר וכתיבת ביטוי מתמטי שאינו גורם לשינוי בערך אחד לפחות היא בדרך כלל שגיאה של המתכנת.

```
class NoResultApp
{
   public static void Main()
   {
      int i;
      int j;

      i + j;   // Error because call is not assigned to a variable.
   }
}
```

רוב האופרטורים עובדים רק עם נתונים מספריים כגון Byte, Short, Long, Integer, Single, Double ו-Decimal. יוצאים מהכלל הם אופרטורי ההשוואה (== ו- =!) ואופרטור ההצבה (=), המסוגלים לעבוד גם על אובייקטים. בנוסף לכך, #C מגדירה את האופרטורים + ו- =+ עבור המחלקה String ואפילו מאפשרת את השימוש באופרטורי ההגדלה (++) וההקטנה (--) על בנאים ייחודיים של השפה כגון נציגים (delegates). בדוגמה האחרונה נעסוק בפרק 14.

# קדימות אופרטורים

כאשר ביטוי יחיד או משפט מכיל אופרטורים מרובים, המהדר חייב לקבוע את הסדר שבו אופרטורים שונים אלה יחושבו. החוקים המכתיבים איך המהדר קובע זאת נקראים **קדימות אופרטורים** (operator precedence). המשמעות של הבנת קדימות אופרטורים יכולה להיות ההבדל שבין כתיבת הביטויים שלך בתחושת ביטחון לבין נעיצת מבט בשורת קוד אחת כאשר אתה תוהה מדוע התוצאה אינה מה שאתה חושב שהיא צריכה להיות.

התבונן בביטוי הבא: 42+6*10. אם אתה מחבר 42 ל- 6 ואז כופל את הסכום ב- 10 התוצאה היא 480. אם אתה כופל 6 ב- 10 ואז מוסיף את המכפלה ל- 42 התוצאה היא 102. במהלך הידור הקוד, חלק מיוחד של המהדר הנקרא **מנתח לקסיקלי** (lexical analyzer) האחראי לקביעה איך צריך לקרוא את הקוד הזה. המנתח הלקסיקלי הוא זה אשר קובע את הקדימות היחסית בין אופרטורים כאשר יותר מסוג אחד של אופרטור נוכח בביטוי. הוא עושה זאת באמצעות הגדרת ערך (או קדימות) לכל אופרטור נתמך. תחילה מחושבים האופרטורים בעלי הקדימות הגבוהה יותר. בדוגמה שלנו, האופרטור * הוא בעל קדימות גבוהה יותר מזו של האופרטור + מאחר והאופרטור * **לוקח** את האופרנדים שלו לפני שהאופרטור + לוקח. הסיבה לכך נובעת מהחוקים הבסיסיים של אריתמטיקה: לכפל ולחילוק יש **תמיד** קדימות גבוהה יותר מזו של חיבור וחיסור. לכן 6 נלקח על ידי ה- * בביטוי 42+6*10 וגם בביטוי 42*6+10 כך שביטויים אלה שקולים ל- 42+(6*10) ו- 10+(42*6).

## קביעת קדימות ב-#C

הבה נראה בצורה מפורשת איך #C מקצה קדימויות לאופרטורים. טבלה 10-1 מציגה את סדר קדימות האופרטורים ב-#C החל מהקדימות הגבוהה ביותר ועד לנמוכה ביותר. בהמשך נדון ביתר פירוט בקטגוריות השונות של אופרטורים הנתמכים ב-#C.

**טבלה 10-1** קדימות אופרטורים

| קטגוריית אופרטור | אופרטורים |
|---|---|
| Primary (בסיסי) | (x), x.y, f(x), a[x], ++x, --x, new, typeof, sizeof, checked, unchecked |
| Unary (אונארי) | +, -, !, ~, ++x, --x, (T)x |
| Multiplicative (כפל וחילוק) | *, /, % |
| Additive (חיבור וחיסור) | +, - |
| Shift (הזזה) | <<, >> |
| Relational (יחסים) | <, >, <=, >=, is |
| Equality (שוויון) | == |
| Logical AND (AND על סיביות) | & |
| Logical XOR (XOR על סיביות) | ^ |
| Logical OR (OR על סיביות) | \| |
| Conditional AND (AND התניית) | && |
| Conditional OR (OR התניית) | \|\| |
| Conditional (התנייה) | ?: |
| Assignment (הצבה) | =, *=, /=, +=, -=, <<=, >>=, &=, ^=, \|= |

## אסוציאטיביות ימנית ושמאלית

אסוציאטיביות קובעת איזה צד של הביטוי צריך להיות מחושב תחילה. לשם דוגמה, תוצאת הביטוי הבא יכולה להיות הסכום 21 או 33 כתלות באסוציאטיביות השמאלית או הימנית של האופרטור - :

42-15-6

האופרטור - מוגדר **אסוציאטיבי-שמאלי** (left-associative) ולכן תחילה מחושב 42-15 ולאחר מכן מופחת 6 מן התוצאה. אם האופרטור - היה מוגדר **אסוציאטיבי-ימני** (right-associative), הביטוי בצד הימני של האופרטור היה מחושב תחילה : 15-6 ולאחר מכן התוצאה היתה מופחתת מ- 42.

כל האופרטורים הבינאריים - כלומר אופרטורים שפועלים על שני אופרנדים - חוץ מאופרטור ההצבה, הם אסוציאטיביים-שמאליים, כלומר הביטוי מחושב משמאל לימין. לכן a+b+c שקול ל- (a+b)+c כאשר a+b מחושב תחילה ואז מתחבר c לתוצאה. אופרטור הצבה ואופרטור התנייה הם אסוציאטיביים-ימניים - הביטוי מחושב מימין לשמאל. במילים אחרות, a=b=c שקול ל- a=(b=c). זה עשוי להטעות מספר מתכנתים שרוצים לכתוב אופרטורים מרובים של הצבות בשורה אחת. נסתכל על הדוגמה הבאה:

```
using System;

class RightAssocApp
{
   public static void Main()
   {
      int a = 1;
      int b = 2;
      int c = 3;

      Console.WriteLine("a={0} b={1} c={2}", a, b, c);
      a = b = c;
      Console.WriteLine("After 'a=b=c': a={0} b={1} c={2}", a, b, c);
   }
}
```

הנה התוצאה המתקבלת מביצוע הדוגמה:

```
a=1 b=2 c=3
After 'a=b=c': a=3 b=3 c=3
```

ביטוי המחושב מימין לשמאל עשוי להיות מבלבל בהתחלה, אך חשוב על זה כך: אם אופרטור ההצבה היה אסוציאטיבי-שמאלי, המהדר היה מחשב תחילה את a=b, ואז a היה מקבל את הערך 2, לאחר מכן הוא היה מחשב את b=c ואז b היה מקבל את הערך 3. התוצאה הסופית היתה a=2 b=3 c=3. כמובן שזו אינה תוצאת השוויון המצופה עבור a=b=c וזו הסיבה שאופרטורי הצבה והתנייה הם אסוציאטיביים-ימניים.

## שימוש מעשי

אין דבר מכשיל יותר מניסיון לאתר שגיאה זמן ממושך, ובסוף לגלות שהשגיאה נגרמה מכך שהמפתח (המתכנת) אינו יודע את החוקים של קדימות או אסוציאטיביות. ברשימות דיוור ראיתי מספר הודעות מאנשים נבונים שהציעו מוסכמת תכנות לפיה תווי רווח מציינים אילו אופרטורים הם **חושבים** שיהיו בעלי קדימות, מעין מנגנון של תיעוד עצמי. כך לדוגמה, מאחר ואנו יודעים שלאופרטור הכפל יש קדימות גבוהה יותר מזו של אופרטור החיבור, נוכל לכתוב קוד בדומה לזה, כאשר הרווחים מגדירים את סדר הקדימות **שאליו מתכוונים**:

```
a =  b*c  +  d;
```

גישה זו לקויה באופן יסודי – המהדר אינו מנתח קוד בצורה נכונה ללא תחביר מוגדר. הוא מנתח ביטויים על סמך החוקים שנקבעו על ידי מפתחי המהדר. לשם כך קיים סימן הסוגריים המאפשר לך להגדיר קדימות ואסוציאטיביות. לדוגמה, ניתן לשכתב את הביטוי a=b*c+d כך: a=(b*c)+d או כך: a=b*(c+d), והמהדר יחשב תחילה את הביטוי שבתוך הסוגריים. אם קיימים שניים או יותר זוגות של סוגריים, המהדר מחשב את הביטוי בכל זוג ולאחר מכן את כל המשפט על פי חוקי הקדימות והאסוציאטיביות שתוארו.

דעתי האיתנה היא שתמיד יש להשתמש בסוגריים כאשר משתמשים באופרטורים מרובים בביטוי יחיד. אני ממליץ לך על כך גם אם אתה יודע את סדר הקדימות כי מי שמתחזק את הקוד שלך עשוי שלא להיות בקי בכך.

# אופרטורים של #C

יש תועלת בחשיבה על אופרטורים בהתאם לקדימותם. בחלק זה אתאר את האופרטורים השכיחים ביותר בסדר שהוצג בטבלה 10-1.

## אופרטורים בסיסיים של ביטויים

הקטגוריה הראשונה שאותה אתאר היא זו של אופרטורים בסיסיים. מאחר ורוב האופרטורים האלה הם די בסיסיים, אפרט אותם ואתן תיאור תמציתי של תפקודם. לאחר מכן אתן הסבר מעט יותר מעמיק עבור אופרטורים שהם פחות מובנים מאליהם.

- **(x)** – צורה זו של אופרטור הסוגריים מאפשרת שליטה על קדימות בפעולות מתמטיות או בקריאה לשיטות.
- **(x.y)** – אופרטור "הנקודה" משמש לתיאור חבר של מחלקה או מבנה (אלמנט), כאשר x מייצג את היישות המכילה ו-y מייצג את החבר.
- **f(x)** – צורה זו של אופרטור הסוגריים משמשת לפירוט הארגומנטים עבור שיטה.
- **a[x]** – צורה זו עם סוגריים מרובעים נקראת אופרטור האינדקס ומאפשרת לגשת לתוך איבר במערך. היא משמשת גם ביחד עם סדרנים שבאמצעותם ניתן להתייחס אל אובייקטים כאל מערכים. למידע נוסף על סדרנים עיין בפרק 7.
- **x++** – אופרטור ההגדלה נסקר בהרחבה בסעיף "אופרטורים של הגדלה והקטנה" בהמשך הפרק.
- **x--** – אופרטור ההקטנה נסקר בהרחבה בהמשך הפרק.
- **new** – האופרטור new משמש ליצירת מופעים (אובייקטים) מהגדרות של מחלקות.

## typeof

**שיקוף** (reflection) הוא היכולת לקבל מידע על סוג נתונים בזמן ריצה. מידע זה כולל את שמות הסוגים, שמות מחלקות וחברי מבנים. חלק אינטגרלי מיכולת זו של מסגרת NET היא מחלקה הנקראת System.Type. מחלקה זו היא המקור עבור כל פעולות השיקוף, וניתן לקבל אותה באמצעות האופרטור **typeof**. נושא השיקוף נסקר בהרחבה בפרק 16, כאן רק נציג דוגמה פשוטה הממחישה כמה קל להשתמש באופרטור typeof בכדי לקבל כמעט כל מידע רצוי אודות סוג או אובייקט בזמן ריצה:

```
using System;
using System.Reflection;

public class Apple
{
   public int nSeeds;
   public void Ripen()
   {
   }
}

public class TypeOfApp
{
   public static void Main()
   {
      Type t = typeof(Apple);
      string className = t.ToString();

      Console.WriteLine("\nInformation about the class {0}",
         className);

      Console.WriteLine("\n{0} methods", className);
      Console.WriteLine("----------------------------");
      MethodInfo[] methods = t.GetMethods();
      foreach (MethodInfo method in methods)
      {
         Console.WriteLine(method.ToString());
      }

      Console.WriteLine("\nAll {0} members", className);
      Console.WriteLine("----------------------------");
      MemberInfo[] allMembers = t.GetMembers();
      foreach (MemberInfo member in allMembers)
      {
         Console.WriteLine(member.ToString());
      }
   }
}
```

התוכנית מכילה מחלקה הנקראת Apple. מחלקה זו מכילה שני חברים בלבד: שדה הנקרא nSeeds ושיטה הנקראת Ripen. תחילה אני משתמש באופרטור typeof ובשם המחלקה בכדי לקבל אובייקט System.Type, שאותו אני מאחסן לאחר מכן במשתנה הנקרא t. מרגע זה ואילך אני יכול להשתמש בשיטות של System.Type בכדי לקבל את כל השיטות והחברים של המחלקה Apple. משימות אלו מתבצעות באמצעות השיטות GetMethods ו-GetMembers, בהתאמה. התוצאות של הקריאה לשיטות אלו מודפסות לפלט הסטנדרטי, כמוצג כאן:

```
Information about the class Apple

Apple methods
-----------------------------
Int32 GetHashCode()
Boolean Equals(System.Object)
System.String ToString()
Void Ripen()
System.Type GetType()

All Apple members
-----------------------------
Int32 nSeeds
Int32 GetHashCode()
Boolean Equals(System.Object)
System.String ToString()
Void Ripen()
System.Type GetType()
Void .ctor()
```

לפני שנמשיך, ברצוני להפנות את תשומת הלב לשתי נקודות. ראשית, רשימת הפלט כוללת גם את חברי המחלקה שעברו בהורשה. מאחר ולא גזרתי בצורה מפורשת את המחלקה ממחלקה אחרת, אנו יודעים שכל החברים שאינם מוגדרים במחלקה Apple עברו בהורשה ממחלקת הבסיס המשתמעת System.Type. שנית, ניתן גם להשתמש בשיטה GetType בכדי לקבל את האובייקט System.Type. שיטה זו, שעוברת בהורשה מ-System.Type , מאפשרת לך לעבוד עם אובייקטים, בניגוד למחלקות. לכן, שני קטעי הקוד הבאים יכולים לשמש באופן תחליפי לקבלת אובייקט System.Type:

```
// Retrieve the System.Type object from the class definition.
Type t1 = typeof(Apple);

// Retrieve the System.Type object from the object.
Apple apple = new Apple();
Type t2 = apple.GetType();
```

## sizeof

האופרטור sizeof משמש לקבלת הגודל בבתים של סוג נתון. יש לזכור שני גורמים חשובים ביותר כאשר משתמשים באופרטור זה. ראשית, ניתן להשתמש באופרטור sizeof רק עם סוגי ערכים (value types). לפיכך, למרות שניתן להשתמש באופרטור זה על חברי מחלקה, לא ניתן להשתמש בו על סוג המחלקה עצמו. שנית, ניתן להשתמש באופרטור זה רק בשיטה או בבלוק קוד המסומן כ-unsafe – נושא שבו נעסוק בפרק 17. הנה דוגמה לשימוש באופרטור sizeof מתוך שיטת מחלקה המסומנת כ-unsafe:

```
using System;

class BasicTypes
{
   // NOTE: You must declare that code that uses the sizeof
   // operator as unsafe.
   static unsafe public void ShowSizes()
   {
      Console.WriteLine("\nBasic type sizes");
      Console.WriteLine("sizeof short = {0}", sizeof(short));
      Console.WriteLine("sizeof int = {0}", sizeof(int));
      Console.WriteLine("sizeof long = {0}", sizeof(long));
      Console.WriteLine("sizeof bool = {0}", sizeof(bool));
   }
}

class Unsafe1App
{
   unsafe public static void Main()
   {
      BasicTypes.ShowSizes();
   }
}
```

הנה תוצאות הביצוע של יישום זה:

```
Basic type sizes
sizeof short = 2
sizeof int = 4
sizeof long = 8
sizeof bool = 1
```

מלבד סוגים מובנים בסיסיים, האופרטור sizeof יכול לשמש לקביעת הגדלים של סוגי ערכים מותאמים אחרים כמו מבנים. עם זאת, התוצאות של האופרטור sizeof יכולות להיות לעיתים פחות ברורות, כמו בדוגמה הבאה:

```
// Using the sizeof operator.
using System;

struct StructWithNoMembers
{
}

struct StructWithMembers
{
   short s;
   int i;
   long l;
   bool b;
}

struct CompositeStruct
{
   StructWithNoMembers a;
   StructWithMembers b;

   StructWithNoMembers c;
}

class UnSafe2App
{
  unsafe public static void Main()
  {
   Console.WriteLine("\nsizeof StructWithNoMembers structure = {0}",
                     sizeof(StructWithNoMembers));
   Console.WriteLine("\nsizeof StructWithMembers structure = {0}",
                     sizeof(StructWithMembers));
   Console.WriteLine("\nsizeof CompositeStruct structure = {0}",
                     sizeof(CompositeStruct));
  }
}
```

אנו מצפים שהיישום ידפיס ערך של 0 עבור המבנה נטול החברים (StructWithNoMembers), ערך של 15 עבור המבנה עם ארבעת החברים הבסיסיים (StructWithMembers), וערך של 15 עבור המבנה המקבץ את השניים (CompositeStruct).

אולם אלה התוצאות המתקבלות בפועל:

```
sizeof StructWithNoMembers structure = 1

sizeof StructWithMembers structure = 16

sizeof CompositeStruct structure = 24
```

הסיבה לכך נעוצה בריפוד ויישור מבנה (padding and structure alignment), המתייחסים לאופן שבו המהדר מסדר struct בתמונת קובץ הפלט. לדוגמה, אם אורך המבנה הוא 3 בתים ויישור הבתים נעשה במרווח של 4 בתים, המהדר מרפד אותו אוטומטית בבית נוסף והאופרטור sizeof ידווח שה-struct הוא באורך 4 בתים. לכן יש לקחת זאת בחשבון בקביעת גודל של מבנה ב-#C.

### מסומן ולא מסומן

שני אופרטורים אלה מאפשרים בקרת גלישה (overflow) מתחום הנתונים בפעולות מתמטיות. מאחר ואופרטורים אלה נוגעים במיוחד לתחום של טיפול בשגיאות, הם נידונים בפרק 12 העוסק בתחום זה.

## אופרטורים מתמטיים

שפת #C תומכת באופרטורים המתמטיים הבסיסיים הנתמכים כמעט בכל שפות התכנות: כפל (*), חילוק (/), חיבור (+), חיסור (-) ושארית (%). ארבעת האופרטורים הראשונים מובנים מאליהם. אופרטור השארית (נקרא גם מודולוס) נותן את השארית של חלוקה במספר שלם. הקוד הבא ממחיש את השימוש באופרטורים אלה:

```
using System;

class MathOpsApp
{
   public static void Main()
   {
      // The System.Random class is part of the .NET
      // Framework class library. Its default constructor
      // seeds the Next method using the current date/time.
      Random rand = new Random();
      int a, b, c;

      a = rand.Next() % 100; // Limit max to 99.
      b = rand.Next() % 100; // Limit max to 99.

      Console.WriteLine("a={0} b={1}", a, b);

      c = a * b;
      Console.WriteLine("a * b = {0}", c);
```

```
        // Note the following code uses integers. Therefore,
        // if a is less than  b, the result will always
        // be 0. To get a more accurate result, you would
        // need to use variables of type double or float.
        c = a / b;
        Console.WriteLine("a / b = {0}", c);

        c = a + b;
        Console.WriteLine("a + b = {0}", c);

        c = a - b;
        Console.WriteLine("a - b = {0}", c);

        c = a % b;
        Console.WriteLine("a % b = {0}", c);
    }
}
```

## אופרטורים אונאריים

קיימים שני אופרטורים אונאריים, פלוס ומינוס. אופרטור מינוס אונארי מורה למהדר שיש להפוך את הסימן של האופרנד, כלומר להפוך אותו לשלילי. לכן בדוגמה הבאה a יקבל את הערך 42- :

```
using System;

class Unary1App
{
    public static void Main()
    {
        int a = 0;

        a = -42;
        Console.WriteLine("{0}", a);
    }
}
```

עם זאת, אי-בהירות נוצרת כאשר כותבים משהו כמו זה :

```
using System;

class Unary2App
{
    public static void Main()
    {
```

```
        int a;
        int b = 2;
        int c = 42;

        a = b * -c;
        Console.WriteLine("{0}", a);
    }
}
```

השורה a=b*-c עשויה להיות מעט מבלבלת. שוב, השימוש בסוגריים עוזר להבהיר שורת קוד זו:

```
// With parentheses, it's obvious that you're multiplying
// b by the negative of c.
a = b * (-c);
```

אם אופרטור מינוס אונארי מחזיר את הערך השלילי של אופרנד, ודאי תחשוב שאופרטור פלוס אונארי מחזיר את הערך החיובי של אופרנד. עם זאת, פלוס אונארי אינו עושה דבר מלבד להחזיר את האופרנד בצורתו המקורית. לכן אין לו כל השפעה על האופרנד. לדוגמה, הפלט של הקוד הבא הוא 84-:

```
using System;

class Unary3App
{
    public static void Main()
    {
        int a;
        int b = 2;
        int c = -42;

        a = b * (+c);
        Console.WriteLine("{0}", a);
    }
}
```

בכדי לקבל את הצורה החיובית של אופרנד, משתמשים בפונקציה המתמטית Math.Abs. הקוד הבא נותן תוצאה של 84:

```
using System;

class Unary4App
{
    public static void Main()
    {
        int a;
        int b = 2;
        int c = -42;
```

```
        a = b * Math.Abs(c);
        Console.WriteLine("{0}", a);
    }
}
```

אופרטור אונארי אחרון אותו אזכיר כאן בקצרה הוא x(T). זו צורה של אופרטור סוגריים המאפשר להמיר סוג אחד לסוג אחר. היות וניתן להעמיס אופרטור זה על ידי הגדרתו מחדש לשימוש עם סוגים מותאמים (יצירת המרה מותאמת), הוא נסקר בפרק 13 העוסק בהעמסת אופרטורים.

## אופרטורים מורכבים של הצבה

**אופרטור הצבה מורכב** (compound assignment operator) הוא צירוף של אופרטור בינארי ואופרטור ההצבה (=). התחביר שלו הוא:

```
lvalue op= vvalue
x  op=  y ;
```

כאשר op מייצג את האופרטור. במקום ש-lvalue (הערך השמאלי) מוחלף ב-rvalue (הערך הימני), לאופרטור המורכב יש אותה השפעה כמו הכתיבה:

```
x = x op y ;
```

על ידי שימוש ב-lvalue כבסיס לתוצאת הפעולה.

שים לב שאני משתמש במילים "יש אותה השפעה". המהדר אינו מתרגם מילולית משהו כמו x+=5 ל- x=x+5. פשוט כך זה עובד מבחינה לוגית. למעשה, יש לנקוט זהירות גדולה כאשר lvalue הוא שיטה. נבחן זאת בעזרת דוגמה:

```
using System;

class CompoundAssignment1App
{
    protected int[] elements;
    public int[] GetArrayElement()
    {
        return elements;
    }

    CompoundAssignment1App()
    {
        elements = new int[1];
        elements[0] = 42;
    }

    public static void Main()
```

```
    {
        CompoundAssignment1App app = new CompoundAssignment1App();

        Console.WriteLine("{0}", app.GetArrayElement()[0]);
        app.GetArrayElement()[0] = app.GetArrayElement()[0] + 5;
        Console.WriteLine("{0}", app.GetArrayElement()[0]);
    }
}
```

שים לב לשורה המודגשת - הקריאה לשיטה CompoundAssignment1App.GetArrayElement והשינוי העוקב באיבר הראשון שלה - שבה השתמשתי בתחביר ההצבה של:

x = x **op** y

הנה קוד MSIL שנוצר מיישום זה:

```
// Inefficient technique of x = x op y.

.method public hidebysig static void Main() il managed
{
  .entrypoint
  // Code size       79 (0x4f)
  .maxstack  4
  .locals (class CompoundAssignment1App V_0)
  IL_0000:  newobj     instance void CompoundAssignment1App::.ctor()
  IL_0005:  stloc.0
  IL_0006:  ldstr      "{0}"
  IL_000b:  ldloc.0
  IL_000c:  call       instance int32[] CompoundAssignment1App::GetArrayElement()
  IL_0011:  ldc.i4.0
  IL_0012:  ldelema    ['mscorlib']System.Int32
  IL_0017:  box        ['mscorlib']System.Int32
  IL_001c:  call       void ['mscorlib']System.Console::WriteLine
(class System.String,
 class System.Object)
  IL_0021:  ldloc.0
  IL_0022:  call
instance int32[] CompoundAssignment1App::GetArrayElement()
  IL_0027:  ldc.i4.0
  IL_0028:  ldloc.0
  IL_0029:  call
instance int32[] CompoundAssignment1App::GetArrayElement()
  IL_002e:  ldc.i4.0
  IL_002f:  ldelem.i4
  IL_0030:  ldc.i4.5
  IL_0031:  add
```

```
  IL_0032:  stelem.i4
  IL_0033:  ldstr      "{0}"
  IL_0038:  ldloc.0
  IL_0039:  call
instance int32[] CompoundAssignment1App::GetArrayElement()
  IL_003e:  ldc.i4.0
  IL_003f:  ldelema    ['mscorlib']System.Int32
  IL_0044:  box        ['mscorlib']System.Int32
  IL_0049:  call  void ['mscorlib']System.Console::WriteLine
                          (class System.String, class System.Object)
  IL_004e:  ret
} // end of method. 'CompoundAssignment1App::Main'
```

השורות המודגשות בקוד MSIL זה מראות כי למעשה פעמיים נקראת השיטה CompoundAssignment1App.GetArrayElement! במקרה הטוב ביותר אין זה נחשב יעיל. במקרה הגרוע ביותר זה יכול להיות איום ונורא, תלוי מה עוד השיטה עושה.

בקוד הבא שיניתי את ההצבה לתחביר אופרטור ההצבה המורכב:

```
using System;

class CompoundAssignment2App
{
   protected int[] elements;
   public int[] GetArrayElement()
   {
      return elements;
   }

   CompoundAssignment2App()
   {
      elements = new int[1];
      elements[0] = 42;
   }

   public static void Main()
   {
      CompoundAssignment2App app = new CompoundAssignment2App();

      Console.WriteLine("{0}", app.GetArrayElement()[0]);
      app.GetArrayElement()[0] += 5;
      Console.WriteLine("{0}", app.GetArrayElement()[0]);
   }
}
```

השימוש באופרטור ההצבה המורכב מביא לקוד MSIL הרבה יותר יעיל. כך זה נראה:

```
// More efficient technique of x op= y.

.method public hidebysig static void Main() il managed
{
  .entrypoint
  // Code size       76 (0x4c)
  .maxstack  4
  .locals (class CompoundAssignment1App V_0, int32[] V_1)
  IL_0000:  newobj     instance void CompoundAssignment1App::.ctor()
  IL_0005:  stloc.0
  IL_0006:  ldstr      "{0}"
  IL_000b:  ldloc.0
  IL_000c:  call instance int32[] CompoundAssignment1App::GetArrayElement()
  IL_0011:  ldc.i4.0
  IL_0012:  ldelema    ['mscorlib']System.Int32
  IL_0017:  box        ['mscorlib']System.Int32
  IL_001c:  call   void ['mscorlib']System.Console::WriteLine
                                (class System.String, class System.Object)
    IL_0021:  ldloc.0
    IL_0022:  call  instance int32[]
CompoundAssignment1App::GetArrayElement()

    IL_0027:  dup
    IL_0028:  stloc.1
    IL_0029:  ldc.i4.0
    IL_002a:  ldloc.1

  IL_002b:  ldc.i4.0
  IL_002c:  ldelem.i4
  IL_002d:  ldc.i4.5
  IL_002e:  add
  IL_002f:  stelem.i4
  IL_0030:  ldstr      "{0}"
  IL_0035:  ldloc.0
  IL_0036:  call    instance int32[] CompoundAssignment1App::GetArrayElement()
  IL_003b:  ldc.i4.0
  IL_003c:  ldelema    ['mscorlib']System.Int32
  IL_0041:  box        ['mscorlib']System.Int32
  IL_0046:  call  void ['mscorlib']System.Console::WriteLine
                                (class System.String, class System.Object)
  IL_004b:  ret
} // end of method 'CompoundAssignment1App::Main'
```

אנו רואים שנעשה שימוש בקוד הפעולה dup.

קוד פעולה זה משכפל את האיבר העליון במחסנית, ובכך יוצר עותק של הערך המתקבל מהקריאה לשיטה CompoundAssignment1App.GetArrayElement.

מטרת תרגיל זה היתה להמחיש, שלמרות שהביטוי x+=y נתפס כשקול לביטוי x=x+y, מתגלים הבדלים דקים בקוד MSIL שנוצר. הבדלים אלה מורים על הצורך להיות זהיר מאד בבחירת התחביר בכל אחת מהנסיבות. כלל אצבע, וזו גם המלצתי, להשתמש באופרטורי הצבה מורכבים בכל זמן שניתן ואפשר.

## אופרטורים של הגדלה והקטנה

אופרטורים של הגדלה הם שרידים של אותם קיצורי דרך אשר הוצגו לראשונה בשפת C והמשיכו גם ב- C++ ו-Java. הם מאפשרים להגדיר בצורה תמצית את הרצון להגדיל או להקטין ב- 1 משתנה המייצג ערך מספרי. לפיכך, i++ שקול להוספת 1 לערך הנוכחי של i.

קיימות שתי גרסאות של האופרטורים להגדלה ולהקטנה והן מהוות פעמים רבות מקור לבלבול. סוג האופרטור, אשר בדרך כלל מתייחסים לפעולתו כאל מוקדמת (prefix) או מאוחרת (postfix), מציין מתי מושפע ערך המשתנה. בגרסאות של אופרטורים להגדלה או הקטנה מוקדמת - כלומר, a++ ו- a-- בהתאמה - תחילה מתבצעת הפעולה ולאחר מכן מתבצעת ההצבה. בגרסאות של אופרטורים להגדלה או הקטנה מאוחרת - כלומר, a++ ו- a-- בהתאמה - תחילה מתבצעת ההצבה ולאחר מכן מתבצעת הפעולה. נתבונן בדוגמה הבאה:

```
using System;

class IncDecApp
{
   public static void Foo(int j)
   {
      Console.WriteLine("IncDecApp.Foo j = {0}", j);
   }

   public static void Main()
   {
      int i = 1;

      Console.WriteLine("Before call to Foo(i++) = {0}", i);
      Foo(i++);
      Console.WriteLine("After call to Foo(i++) = {0}", i);

      Console.WriteLine("\n");

      Console.WriteLine("Before call to Foo(++i) = {0}", i);
      Foo(++i);
      Console.WriteLine("After call to Foo(++i) = {0}", i);
   }
```

```
}
```

יישום זה מפיק את הפלט הבא:

```
Before call to Foo(i++) = 1
IncDecApp.Foo j = 1
After call to Foo(i++) = 2

Before call to Foo(++i) = 2
IncDecApp.Foo j = 3
After call to Foo(++i) = 3
```

כפי שרואים, ההבדל הוא **מתי** הערך נוצר והאופרנד משתנה. בקריאה ל-Foo(i++), הערך i מועבר (ללא שינוי) ל-Foo, ולאחר שהשיטה Foo חוזרת, מגדילים את i ב- 1. תוכל לראות זאת בקטע MSIL שלהלן. שים לב שקוד הפעולה add אינו נקרא עד **לאחר** שהערך מוצב במחסנית.

```
IL_0013:  ldloc.0
IL_0014:  dup
IL_0015:  ldc.i4.1
IL_0016:  add
IL_0017:  stloc.0
IL_0018:  call      void IncDecApp::Foo(int32)
```

כעת נתבונן באופרטור ההגדלה המוקדמת בקריאה ל-Foo(++i). במקרה זה, קוד MSIL המתקבל נראה כמוצג להלן. שים לב שקוד הפעולה add נקרא **לפני** שהערך מוצב במחסנית עבור הקריאה העוקבת לשיטה Foo.

```
IL_0049:  ldloc.0
IL_004a:  ldc.i4.1
IL_004b:  add
IL_004c:  dup
IL_004d:  stloc.0
IL_004e:  call      void IncDecApp::Foo(int32)
```

## אופרטורים של יחסים

רוב האופרטורים מחזירים תוצאה מספרית. אופרטורים של יחסים שונים מעט בכך שהם מייצרים ערך בוליאני. במקום לבצע פעולה מתמטית מסוימת על קבוצה של אופרנדים, אופרטור יחסים משווה את הקשר בין האופרנדים ומחזיר ערך של true אם הקשר הוא אמת, או false אם הקשר אינו אמת.

## אופרטורים של השוואה

קבוצת האופרטורים של יחסים, הנקראים אופרטורים של השוואה, כוללת את קטן מ- (<), קטן מ- או שווה ל- (=<), גדול מ- (<), גדול מ- או שווה ל- (=<), שווה ל- (==) ושונה מ- (=!). אופרטורים אלה מובנים מאליהם כל זמן שעובדים עם מספרים, אך האופן שבו כל אופרטור פועל על אובייקטים אינו מובן מאליו. הנה דוגמה:

```
using System;

class NumericTest
{
   public NumericTest(int i)
   {
      this.i = i;
   }

   protected int i;
}

class RelationalOps1App
{
   public static void Main()
   {
      NumericTest test1 = new NumericTest(42);
      NumericTest test2 = new NumericTest(42);

      Console.WriteLine("{0}", test1 == test2);
   }
}
```

אם אתה מתכנת Java, אתה יודע מה עומד לקרות כאן. עם זאת, רוב מפתחי C++ קרוב לוודאי יופתעו לראות שדוגמה זו מדפיסה את הערך false. זכור שכאשר אתה מממש אובייקט, אתה מקבל ייחוס לאובייקט שמוקצה במאגר זיכרון. לפיכך, כאשר אתה משתמש באופרטור של יחסים בכדי להשוות בין שני אובייקטים, מהדר C# אינו משווה את תוכנם של האובייקטים אלא את כתובותיהם. שוב, בכדי להבין באופן מושלם מה קורה כאן, נתבונן ב-MSIL עבור קוד זה:

```
.method public hidebysig static void Main() il managed
{
  .entrypoint
  // Code size       39 (0x27)
  .maxstack  3
  .locals (class NumericTest V_0,
           class NumericTest V_1,
           bool V_2)
```

```
  IL_0000:  ldc.i4.s   42
  IL_0002:  newobj     instance void NumericTest::.ctor(int32)
  IL_0007:  stloc.0
  IL_0008:  ldc.i4.s   42
  IL_000a:  newobj     instance void NumericTest::.ctor(int32)
  IL_000f:  stloc.1
  IL_0010:  ldstr      "{0}"
  IL_0015:  ldloc.0
  IL_0016:  ldloc.1
  IL_0017:  ceq
  IL_0019:  stloc.2
  IL_001a:  ldloca.s   V_2
  IL_001c:  box        ['mscorlib']System.Boolean
  IL_0021:  call   void ['mscorlib']System.Console::WriteLine
                                (class System.String,class System.Object)
  IL_0026:  ret
} // end of method 'RelationalOps1App::Main'
```

בשורת locals המהדר מצהיר ששיטת Main זו תכיל שלושה משתנים מקומיים. שני הראשונים הם אובייקטים של NumericTest והשלישי הוא סוג בוליאני. כעת נדלג לשורות IL_0002 ו-IL_0007. כאן מממש MSIL את האובייקט test1, ועם קוד הפעולה stloc הוא מאחסן את הייחוס המוחזר במשתנה המקומי הראשון. עם זאת, נקודת המפתח כאן היא ש-MSIL מאחסן את הכתובת של האובייקט החדש שנוצר. לאחר מכן, קוד IL בשורות IL_000a ו-IL_000f יוצר את האובייקט test2 ומאחסן את הייחוס המוחזר במשתנה המקומי השני. לבסוף, שורות IL_0015 ו-IL_0016 פשוט טוענות את המשתנים המקומיים למחסנית באמצעות הקריאה ל-ldloc, ואז שורה IL_0017 קוראת לקוד פעולה ceq, המשווה את שני הערכים העליונים במחסנית (הייחוסים ל-test1 ול-test2). הערך המוחזר מאוחסן לאחר מכן במשתנה המקומי השלישי ומאוחר יותר הוא מודפס באמצעות הקריאה ל- System.Console.WriteLine.

איך ניתן להשוות בין אובייקטים על בסיס של חבר-אחר-חבר? התשובה נמצאת במחלקת הבסיס המשתמעת של כל האובייקטים במסגרת NET. המחלקה System.Object מכילה שיטה הנקראת Equals המיועדת למטרה זו בדיוק. לדוגמה, הקוד הבא מבצע השוואה בין התכולה של שני האובייקטים כפי שהיינו מצפים ומחזיר את הערך true:

```
using System;

class RelationalOps2App
{
   public static void Main()
   {
      Decimal test1 = new Decimal(42);
      Decimal test2 = new Decimal(42);

      Console.WriteLine("{0}", test1.Equals(test2));
   }
}
```

שים לב שהדוגמה RelationalOps1App משתמשת במחלקה מתוצרת עצמית (NumericTest), והדוגמה השנייה משתמשת במחלקה של NET (Decimal). זאת משום שחייבים לדרוס את השיטה System.Object.Equals בכדי לבצע את ההשוואה של חבר-אחר-חבר. לכן השימוש בשיטה Equals על המחלקה NumericTest לא יעבוד מאחר ולא דרסנו את השיטה. עם זאת, היות והמחלקה Decimal כן דורסת את השיטה Equals שעברה בהורשה, היא אכן מבצעת את המצופה.

דרך אחרת לטפל בהשוואת אובייקטים היא באמצעות העמסת אופרטורים. העמסת אופרטור מגדירה את הפעולות שיש לבצע בין אובייקטים מסוג מסוים. לדוגמה, באובייקטים מסוג String, האופרטור + משרשר את המחרוזות במקום לבצע פעולת חיבור. העמסת אופרטורים נידונה בפרק 13.

## אופרטורים פשוטים של הצבה

הערך הנמצא בצד השמאלי של אופרטור ההצבה נקרא lvalue; הערך בצד הימני נקרא rvalue. rvalue יכול להיות כל קבוע, משתנה, מספר או ביטוי שניתן לתרגם אותו לערך התואם ל-lvalue. עם זאת, lvalue חייב להיות משתנה מסוג מוגדר. הסיבה לכך היא שערך מועתק מימין לשמאל. לפיכך, חייב להיות מקום פיסי מוקצה בזיכרון, המהווה יעד אולטימטיבי עבור הערך החדש. לדוגמה, אפשר לכתוב i=4 כי i מייצג מיקום פיסי בזיכרון, במחסנית או במאגר זיכרון מיוחד, תלוי בסוג בפועל של המשתנה i. עם זאת, לא ניתן לבצע את המשפט 4=i כי 4 הוא ערך, לא משתנה בזיכרון אשר ניתן לשנות את תוכנו. אגב, החוק ב-#C הוא ש-lvalue יכול להיות משתנה, מאפיין או סדרן. למידע נוסף על מאפיינים וסדרנים עיין שוב בפרק 7. בכדי להשאיר את הדברים פשוטים, אשתמש בדוגמאות בפרק רק במשתנים.

למרות שהצבה מספרית היא ברורה למדי, פעולות הצבה עם אובייקטים הן סוגיה מתוחכמת יותר. זכור שכאשר מתעסקים עם אובייקטים, אין מתעסקים עם אלמנטים פשוטים שמוקצים במחסנית ואשר מעתיקים ומעבירים אותם בקלות. כאשר מבצעים מניפולציה על אובייקטים, יש למעשה רק ייחוס ליישות שהוקצה לה מקום במאגר זיכרון. לפיכך, כאשר מנסים להקצות אובייקט (או סוג ייחוס) במשתנה, אין מעתיקים נתונים כפי שעושים עם סוגי ערכים. פשוט מעתיקים ייחוס ממקום אחד למקום אחר.

נניח שיש לך שני אובייקטים: test1 ו-test2. אם אתה מצהיר test1=test2, test1 אינו עותק של test2 - הוא בדיוק אותו הדבר! test1 מצביע לאותו מקום בזיכרון כמו test2. לפיכך, כל שינוי באובייקט test1 הוא גם שינוי באובייקט test2.

הנה תוכנית הממחישה זאת:

```
using System;

class Foo
{
   public int i;
}

class RefTest1App
{
   public static void Main()
   {
      Foo test1 = new Foo();
      test1.i = 1;

      Foo test2 = new Foo();
      test2.i = 2;

      Console.WriteLine("BEFORE OBJECT ASSIGNMENT");
      Console.WriteLine("test1.i={0}", test1.i);
      Console.WriteLine("test2.i={0}", test2.i);
      Console.WriteLine("\n");

      test1 = test2;

      Console.WriteLine("AFTER OBJECT ASSIGNMENT");
      Console.WriteLine("test1.i={0}", test1.i);
      Console.WriteLine("test2.i={0}", test2.i);
      Console.WriteLine("\n");

      test1.i = 42;

      Console.WriteLine("AFTER CHANGE TO ONLY TEST1 MEMBER");
      Console.WriteLine("test1.i={0}", test1.i);
      Console.WriteLine("test2.i={0}", test2.i);
      Console.WriteLine("\n");
   }
}
```

אם תבצע את הקוד תקבל את הפלט הבא:

```
BEFORE OBJECT ASSIGNMENT
test1.i=1
test2.i=2

AFTER OBJECT ASSIGNMENT
test1.i=2
test2.i=2

AFTER CHANGE TO ONLY TEST1 MEMBER
test1.i=42
test2.i=42
```

הבה נעבור על הדוגמה צעד אחר צעד בכדי להבין מה קורה כאן. Foo היא מחלקה פשוטה המגדירה חבר יחיד בשם i. שני מופעים של מחלקה זו - test1 ו-test2 - נוצרים בשיטה Main כאשר בכל מקרה מגדירים ערך לחבר i של האובייקט החדש (ערך של 1 עבור הראשון וערך של 2 עבור השני). בנקודה זו אנו מדפיסים את הערכים והם נראים כפי שהיינו מצפים, כאשר test1.i הוא 1 ו-test2.i הוא 2. מכאן מתחיל להיות משעשע. השורה הבאה מקצה את אובייקט test2 באובייקט test1. מתכנתי Java הקוראים שורות אלו יודעים מה צפוי. עם זאת, רוב מתכנתי ++C היו מצפים שהחבר i של האובייקט test1 יהיה שווה כעת לחברים של האובייקט test2 (מתוך הנחה שאם היישום עובר שם הידור, חייב להיות מבוצע סוג מסוים של אופרטור העתקה משתמע כלפי החבר). למעשה, כך זה נראה בהדפסת הערך של חברי שני האובייקטים. עם זאת, הקשר בין אובייקטים אלו הוא כעת הרבה יותר עמוק מזה. הקוד מציב ערך של 42 ב-test1.i ושוב מדפיס את ערכי i של שני האובייקטים. אנו רואים שהשינוי באובייקט test1 שינה את אובייקט test2 גם כן. זאת משום שהאובייקט שהיה ידוע מקודם כ-test1 אינו קיים עוד. בהקצאה שבוצעה, אובייקט test1 אבד באופן בסיסי מאחר ואין אליו יותר התייחסות ביישום, ובסופו של דבר הוא נאסף באמצעות garbage collector. האובייקטים test1 ו-test2 מצביעים כעת לאותו מקום בזיכרון. לכן כל שינוי שנעשה באחד מהמשתנים משתקף במשתנה האחר.

שים לב בשתי השורות האחרונות של הפלט כי למרות שהקוד קובע ערך רק ל-test1.i, גם הערך של test2.i מושפע. שוב, זה קורה מאחר ושני המשתנים מצביעים כעת לאותו מקום בזיכרון, וזו גם ההתנהגות שלה היית מצפה אם היית מתכנת ב-Java. אולם, זה עומד בניגוד מוחלט למה שמפתח ב-++C היה מצפה, כי ב-++C משמעות הפעולה של העתקת אובייקטים היא שכל משתנה פשוט מקבל עותק משלו של החברים כך שלשינויים באובייקט אחד אין כל השפעה על האחר.

מאחר וחשוב מאד להבין איך לעבוד עם אובייקטים ב-#C, הבה נסתכל מה קורה כאשר מעבירים אובייקט לשיטה:

```
using System;

class Foo
{
   public int i;
}

class RefTest2App
{
   public void ChangeValue(Foo f)
   {
      f.i = 42;
   }

   public static void Main()
   {
      RefTest2App app = new RefTest2App();

      Foo test = new Foo();
      test.i = 6;

      Console.WriteLine("BEFORE METHOD CALL");
      Console.WriteLine("test.i={0}", test.i);
      Console.WriteLine("\n");

      app.ChangeValue(test);

      Console.WriteLine("AFTER METHOD CALL");
      Console.WriteLine("test.i={0}", test.i);
      Console.WriteLine("\n");
   }
}
```

ברוב השפות - חוץ מ-Java - קוד זה היה יוצר עותק של האובייקט test במחסנית המקומית של השיטה RefTest2App.ChangeValue. אם זה היה המקרה, האובייקט test שנוצר בשיטה Main מעולם לא היה רואה את השינויים שנעשו באובייקט f בתוך השיטה ChangeValue. עם זאת, שוב, מה שקורה כאן הוא שהשיטה Main העבירה ייחוס לאובייקט test שהוקצה לו מקום בזיכרון. כאשר השיטה ChangeValue משנה את המשתנה המקומי f.i שלה, היא גם משנה ישירות את האובייקט test של השיטה Main.

# סיכום

חלק חשוב בכל שפת תכנות הוא הדרך שבה היא מטפלת בפעולות של הצבה, מתמטיקה, יחסים ולוגיקה בכדי לבצע את העבודה הבסיסית הנדרשת על ידי כל יישום של העולם האמיתי. פעולות אלו נשלטות בקוד באמצעות אופרטורים. גורמים שקובעים את ההשפעות שיש לאופרטורים בקוד כוללים קדימות ואסוציאטיביות שמאלית וימנית. #C לא רק מספקת קבוצה רבת עוצמה של אופרטורים מוגדרים מראש, אלא אף מרחיבה אופרטורים אלו באמצעות יישומים מותאמים, בהם נעסוק בפרק 13.

# פרק 11 בקרת זרימה

המשפטים המאפשרים בקרה על זרימת התוכנית ביישום #C מתחלקים לשלוש קטגוריות ראשיות: משפטי בחירה, משפטי איטרציה ומשפטי דילוג. בכל אחד מאלה מתבצע מבחן שתוצאתו היא ערך בוליאני. באמצעות ערך זה שולטים בזרימת הביצוע של היישום. בפרק זה תלמד איך להשתמש בכל אחד מסוגי משפטים אלה בבקרת זרימה.

## משפטי בחירה

משתמשים במשפטי בחירה בכדי לקבוע איזה קוד צריך להתבצע ומתי. #C מציגה שני משפטי בחירה: משפט switch, המאפשר בחירת קוד לביצוע על סמך ערך, ומשפט if המבצע קוד על סמך תנאי בוליאני. משפט if הוא הנפוץ ביותר מבין משפטים אלה.

### משפט if

משפט **if** מבצע אחד או יותר משפטים אם הביטוי המחושב הוא true. להלן תחביר משפט if, כאשר הסוגריים המרובעים מציינים את השימוש האופציונלי בפסוקית else (בה נדון בקרוב):

```
if (expression)
   statement1
[else
   statement2]
```

expression הוא כל ביטוי המחזיר ערך בוליאני. אם הערך המוחזר הוא true, הבקרה מועברת ל-statement1. אם התוצאה היא false וקיימת פסוקית else, הבקרה מועברת ל-statement2. statement1 ו-statement2 יכולים להכיל משפט אחד המסתיים בנקודה ופסיק (נקרא משפט פשוט) או משפטים מרובים המוקפים בסוגריים מסולסלים

(משפט מורכב). הדוגמה הבאה ממחישה משפט מורכב המתבצע אם expression1 נותן תוצאה true:

```
if (expression1)
{
   statement1
   statement2
}
```

בדוגמה הבאה, המשתמש מתבקש להקליד מספר בין 1 ל- 10. לאחר מכן נוצר מספר אקראי, ולמשתמש נאמר אם המספר שהוא ניחש תואם למספר האקראי. דוגמה פשוטה זו ממחישה איך להשתמש במשפט if ב-C#:

```
using System;

class IfTest1App
{
   const int MAX = 10;

   public static void Main()
   {
      Console.Write("Guess a number between 1 and {0}...", MAX);
      string inputString = Console.ReadLine();

      int userGuess = inputString.ToInt32();

      Random rnd = new Random();
      double correctNumber = rnd.NextDouble() * MAX;
      correctNumber = Math.Round(correctNumber);

      Console.Write("The correct number was {0} and you guessed {1}...",
                    correctNumber, userGuess);
      if (userGuess == correctNumber) // They got it right!
      {
         Console.WriteLine("Congratulations!");
      }
      else // Wrong answer!
      {
         Console.WriteLine("Maybe next time!");
      }
   }
}
```

## ריבוי תנאים במשפט if

פסוקית **else** של משפט if מאפשרת להגדיר מסלול חלופי של פעולה אם תוצאת מבחן if היא false. בדוגמת ניחוש המספרים, היישום ביצע השוואה פשוטה בין המספר שהמשתמש ניחש לבין מספר שחולל באופן אקראי. במקרה זה קיימות רק שתי אפשרויות: המשתמש ניחש נכון או לא. ניתן לשלב את if ו-else כך שיטפלו במצבים בהם רוצים לבחון יותר משני תנאים. בדוגמה שלהלן מתבקש המשתמש לציין באיזו שפה הוא כותב כרגע (חוץ מ-C#). כאן הוספתי את היכולת לבחור מתוך שלוש שפות, כך שהמשפט if חייב להיות מסוגל לטפל בארבע תשובות אפשריות: שלוש השפות והמקרה שבו המשתמש בוחר בשפה לא ידועה. הנה הדרך לתכנת זאת באמצעות משפט if/else:

```
using System;

class IfTest2App
{
   const string CPlusPlus = "C++";
   const string VisualBasic = "Visual Basic";
   const string Java = "Java";

   public static void Main()
   {
      Console.Write("What is your current language of choice " + "(excluding C#)?");
      string inputString = Console.ReadLine();

      if (0 == String.Compare(inputString, CPlusPlus, true))
      {
         Console.WriteLine("\nYou'll have no problem picking " + "up C# !");
      }
      else if (0 == String.Compare(inputString, VisualBasic, true))
      {
         Console.WriteLine("\nYou'll find lots of cool VB features " + "in C# !");
      }
      else if (0 == String.Compare(inputString, Java, true))
      {
         Console.WriteLine("\nYou'll have an easier time " + "picking up C# <G> !!");
      }
      else
      {
         Console.WriteLine("\nSorry - doesn't compute.");
      }
   }
}
```

שים לב לשימוש באופרטור == הבא בכדי לבצע השוואה בין 0 לערך המוחזר של String.Compare. זאת מכיוון ש-String.Compare מחזירה 1- אם המחרוזת הראשונה קטנה מהשנייה, 1 אם המחרוזת הראשונה גדולה מהשנייה, ו- 0 אם שתי המחרוזות זהות. עם זאת, דבר זה ממחיש מספר פרטים מעניינים אודות האופן שבו #C אוכפת את השימוש במשפט if, כמתואר בסעיף הבא.

## אכיפת חוקי if ב-#C

היבט אחד המפתיע מתכנתי #C חדשים היא העובדה שתוצאת הביטוי המחושב חייבת להיות ערך בוליאני. זאת בניגוד לשפות כמו ++C שבהן ניתן להשתמש במשפט if בכדי לבחון כל משתנה בעל ערך שונה מ- 0. הדוגמה הבאה ממחישה מספר שגיאות נפוצות של מתכנתי ++C כאשר הם מנסים להשתמש במשפט if בשפת #C בפעם הראשונה:

```
using System;

interface ITest
{
}

class TestClass : ITest
{
}

class InvalidIfApp
{
    protected static TestClass GetTestClass()
    {
        return new TestClass();
    }

    public static void Main()
    {
        int foo = 1;
        if (foo) // ERROR: attempting to convert int to bool.
        {
        }

        TestClass t = GetTestClass();
        if (t) // ERROR: attempting to convert TestClass to bool.
        {
            Console.WriteLine("{0}", t);
```

```
        ITest i = t as ITest;
          if (i) // ERROR: attempting to convert ITest to bool.
          {
             // ITest methods
          }
      }
   }
}
```

אם תנסה להדר קוד זה תקבל את שגיאות ההידור הבאות:

```
invalidIf.cs(22,7): error CS0029: Cannot implicitly convert type 'int' to 'bool'
invalidIf.cs(27,7): error CS0029: Cannot implicitly convert type 'TestClass' to 'bool'
invalidIf.cs(31,14): warning CS0183:
The given expression is always of the provided ('ITest') type
invalidIf.cs(32,8): error CS0029: Cannot implicitly convert type 'ITest' to 'bool'
```

כפי שאתה רואה, המהדר פולט שלוש פעמים שגיאה בתגובה לשלושה ניסיונות להשתמש במשפט if עם פעולה שאינה מניבה ערך בוליאני. זאת מכיון שמתכנני C# רצו לעזור במניעת אי-בהירות בקוד והאמינו שהמהדר צריך לאכוף את המטרה המקורית של משפט if - כלומר, בקרה של זרימת הביצוע על סמך התוצאה של מבחן בוליאני. הדוגמה לעיל נכתבה מחדש וכאן ההידור נעשה ללא שגיאות. כל שורה שגרמה לשגיאת מהדר בתוכנית הקודמת שונתה כך שתכיל ביטוי המחזיר תוצאה בוליאנית המספקת את המהדר.

```
using System;

interface ITest
{
}

class TestClass : ITest
{
}

class ValidIfApp
{
   protected static TestClass GetTestClass()
   {
      return new TestClass();
   }
```

```
    public static void Main()
    {
        int foo = 1;
        if (foo > 0)

        {
        }

        TestClass t = GetTestClass();
        if (t != null)
        {
            Console.WriteLine("{0}", t);

            ITest i = t as ITest;
            if (i != null)
            {
                // ITest methods.
            }
        }
    }
}
```

## משפט switch

באמצעות משפט **switch** ניתן להגדיר ביטוי המחזיר ערך אינטגרלי וקטע קוד אחד או יותר שיבוצעו כתלות בתוצאת הביטוי. יש כאן דמיון לשימוש במשפט if/else, אך למרות שניתן להגדיר משפטי תנאי מרובים (שקרוב לוודאי אינם קשורים זה לזה) עם משפטי if/else מרובים, משפט switch מכיל משפט תנאי אחד ולאחריו כל התוצאות שהקוד מוכן לטפל בהם. הנה התחביר:

```
switch (switch_expression)
{
        case constant-expression:
            statement
            jump-statement
        .
        .
        .
        case constant-expressionN:
            statementN
        [default]
}
```

יש לזכור כאן שני כללים עיקריים: תחילה, switch_expression חייב להיות מהסוג (או ניתן להמרה באופן משתמע לסוג) sbyte, byte, short, ushort, int, uint, long, ulong, char

או string (או enum המבוסס על אחד מסוגים אלה). שנית, עליך לספק jump-statement עבור כל משפט case אלא אם זה משפט ה-case האחרון ב-switch, כולל המשפט break. מאחר וכאן זה פועל באופן שונה ממספר שפות אחרות, אסביר נקודה זו ביתר פירוט מאוחר יותר בסעיף "מניעת גלישה במשפטי switch".

משפט switch פועל מבחינה מושגית כמו משפט if. תחילה מחושב הביטוי switch_expression ולאחר מכן התוצאה מושווית לכל אחד מ- constant-expression או תוויות המקרים (case labels), המוגדרים במשפטי case השונים. אם נמצא מקרה המקיים את התנאי, הבקרה מועברת לשורת הקוד הראשונה באותו משפט case.

בנוסף לאפשרות להגדיר משפטי case שונים, משפט switch מאפשר גם להגדיר משפט default שהוא מקרה ברירת המחדל המתאים לכל ערך שאינו ברשימה. זה דומה לפסוקית else של משפט if. שים לב שיכולה להיות רק תווית default אחת עבור כל משפט switch. בנוסף לכך, אם לא נמצאה תווית מתאימה ל- switch_expression, הבקרה מועברת לשורת הקוד הראשונה לאחר הסוגר המסולסל המסיים של המשפט switch. הנה דוגמה שבה מחלקה בשם Payment משתמשת במשפט switch בכדי לקבוע איזה אמצעי תשלום נבחר:

```
using System;

enum Tenders : int
{
   Cash = 1,
   Visa,
   MasterCard,
   AmericanExpress
};

class Payment
{
   public Payment(Tenders tender)
   {
      this.Tender = tender;
   }

   protected Tenders tender;
   public Tenders Tender
   {
      get
      {
         return this.tender;
      }
      set
      {
         this.tender = value;
      }
   }
```

```
    public void ProcessPayment()

    {
        switch ((int)(this.tender))
        {
            case (int)Tenders.Cash:
                Console.WriteLine("\nCash - Accepted");
                break;

            case (int)Tenders.Visa:
                Console.WriteLine("\nVisa - Accepted");
                break;

            case (int)Tenders.MasterCard:
                Console.WriteLine("\nMastercard - Accepted");
                break;

            case (int)Tenders.AmericanExpress:
                Console.WriteLine("\nAmerican Express - Accepted");
                break;

            default:
                Console.WriteLine("\nSorry - Invalid tender");
                break;
        }
    }
}

class SwitchApp
{
    public static void Main()
    {
        Payment payment = new Payment(Tenders.Visa);
        payment.ProcessPayment();
    }
}
```

אנו יוצרים מופע מהמחלקה Payment עם הערך Tenders.Visa ולכן ביצוע היישום מדפיס את הפלט הבא:

```
Visa - Accepted.
```

## צירוף תוויות case

בדוגמה Payment השתמשנו בתווית case שונה לכל אפשרות של השדה Payment.tender. עם זאת, איך ניתן לצרף תוויות case? לדוגמה, אולי תרצה להציג תיבת דו-שיח לאישור כרטיס אשראי עבור כל אחד מסוגי כרטיסי האשראי המפורטים ב- Tenders enum.

במקרה זה תוכל להציב תוויות case אחת אחרי השנייה, באופן כזה:

```
using System;

enum Tenders : int
{
   Cash = 1,
   Visa,
   MasterCard,
   AmericanExpress
};

class Payment
{
   public Payment(Tenders tender)
   {
      this.Tender = tender;
   }

   protected Tenders tender;
   public Tenders Tender
   {
      get
      {
         return this.tender;
      }
      set
      {
         this.tender = value;
      }
   }

   public void ProcessPayment()
   {
      switch ((int)(this.tender))
      {
         case (int)Tenders.Cash:
            Console.WriteLine("\nCash - Everyone's favorite tender.");
            break;

         case (int)Tenders.Visa:
         case (int)Tenders.MasterCard:
         case (int)Tenders.AmericanExpress:
            Console.WriteLine("\nDisplay Credit Card Authorization Dialog.");
            break;
```

```
            default:
                Console.WriteLine("\nSorry - Invalid tender.");
                break;
        }
    }
}

class CombiningCaseLabelsApp
{

    public static void Main()
    {
        Payment payment = new Payment(Tenders.MasterCard);
        payment.ProcessPayment();
    }
}
```

אם אתה יוצר מופע מהמחלקה Payment עם Tenders.Visa, Tenders.MasterCard או Tenders.AmericanExpress, תקבל את הפלט הזה:

Display Credit Card Authorization Dialog.

## מניעת גלישה במשפטי switch

במהלך הפיתוח של #C הכירו מתכנני השפה בצורך ליישם מבחן של סיכון מול תועלת כאשר היה צריך להחליט באם יש לכלול תכונה בשפה. **גלישה** (fall-through) היא דוגמה לתכונה שלא עברה מבחן זה. באופן נורמלי ב- ++C, משפט case מתבצע כאשר constant-expression שלו תואם ל- switch_expression. לאחר מכן מתבצעת יציאה ממשפט switch עם משפט break. גלישה גורמת לביצוע משפט ה-case הבא במשפט switch בהיעדר משפט break.

למרות שגלישות אינן נתמכות ב-#C, הן משמשות בדרך כלל במצבים בהם יש שתי תוויות case והתווית השנייה מייצגת פעולה שיש לבצעה בכל אחד מהמקרים. לדוגמה, פעם כתבתי עורך מסד נתונים ב- ++C המאפשר למשתמשי הקצה ליצור את הטבלאות והשדות שלהם בצורה גרפית. כל אחת מטבלאות אלו הוצגה במבט עץ בממשק משתמש הדומה לסייר Windows. אם המשתמש לחץ על מבט העץ עם הלחצן הימני של העכבר, הצגתי תפריט אפשרויות כמו 'הדפס את כל הטבלאות' או 'צור טבלה חדשה'. אם משתמש הקצה לחץ לחיצה ימנית על טבלה מסוימת, הצגתי תפריט הקשר עבור טבלה זו. עם זאת, תפריט זה היה צריך לכלול גם את כל אפשרויות מבט העץ.

מבחינה מושגית, הקוד שלי היה דומה לקוד הבא:

```
// Dynamically create menu in C++.
switch(itemSelected)
{
   case TABLE:
   // Add menu options based on current table;
   // break left out intentionally.

   case TREE_VIEW:
   // Add menu options for tree view.
   break;
}
// Display menu.
```

תווית case הראשונה מסתכמת לצירוף של שתי התוויות מבלי צורך לשכפל קוד או לשבץ שתי קריאות לאותה שיטה. אולם, מתכנני שפת #C החליטו כי למרות שתכונה זו עשויה להיות נוחה, התועלת שבה אינה שווה את הסיכון הכרוך בכך מאחר ורוב הזמן משמיטים משפטי break שלא במתכוון, דבר הגורר שגיאות תוכנה קשות לאיתור. בכדי לבצע משימה זו ב-#C, מצבך יהיה טוב יותר אם תשתמש במשפט if במקום switch:

```
// Dynamically create menu.
if (itemSelected == TABLE)
{
   // Add menu options based on current table.
}
// Add menu options for tree view.
// Display menu.
```

# משפטי איטרציה

בשפת #C, משפטי while, do/while, for ו-foreach מאפשרים לבצע איטרציה או לולאה מבוקרת. בכל אחד מאלה מתבצע משפט מסוים, פשוט או מורכב, כל עוד ביטוי בוליאני מקבל את הערך true, חוץ מהמקרה של משפט foreach, המשמש לביצוע איטרציה על רשימת אובייקטים.

## משפט while

משפט **while** הוא בעל התחביר הבא:

```
while (Boolean-expression)
embedded-statement
```

אם נשתמש בדוגמת ניחוש המספרים הקודמת, נוכל לשכתב אותה עם משפט while כך שניתן להמשיך את המשחק עד אשר מנחשים את המספר הנכון או עד שמחליטים לפרוש מהמשחק:

```
using System;

class WhileApp
{
   const int MIN = 1;
   const int MAX = 10;
   const string QUIT_CHAR = "Q";

   public static void Main()
   {
      Random rnd = new Random();
      double correctNumber;

      string inputString;
      int userGuess;

      bool correctGuess = false;
      bool userQuit = false;

      while (!correctGuess && !userQuit)
      {
         correctNumber = rnd.NextDouble() * MAX;
         correctNumber = Math.Round(correctNumber);

         Console.Write
            ("Guess a number between {0} and {1}...({2} to quit)",
            MIN, MAX, QUIT_CHAR);
         inputString = Console.ReadLine();

         if (0 == string.Compare(inputString, QUIT_CHAR, true))
            userQuit = true;
         else
         {
            userGuess = inputString.ToInt32();
            correctGuess = (userGuess == correctNumber);

            Console.WriteLine
               ("The correct number was {0}\n",
                correctNumber);
         }
      }
```

```
        if (correctGuess && !userQuit)
        {
            Console.WriteLine("Congratulations!");
        }
        else

        {
            Console.WriteLine("Maybe next time!");
        }
    }
}
```

קידוד וביצוע היישום מפיק פלט בדומה לזה:

```
C:\>WhileApp
Guess a number between 1 and 10...(Q to quit)3
The correct number was 5

Guess a number between 1 and 10...(Q to quit)5
The correct number was 5

Congratulations!

C:\>WhileApp
Guess a number between 1 and 10...(Q to quit)q
Maybe next time!
```

## משפט do/while

בתחביר של המשפט while ניתן להבחין באפשרות לבעיה אחת. ביטוי התנאי (Boolean-expression) מחושב לפני הביצוע של המשפטים בתוך הלולאה (embedded-statement). לכן היישום הקודם מאתחל את המשתנים correctGuess ו-userQuit עם ערך false בכדי להבטיח כניסה ללולאת while. מרגע זה נשלטים ערכים אלה על פי ניחוש המשתמש או רצונו לפרוש. אולם, אם רוצים להבטיח ש-embedded-statement תמיד יבוצע לפחות פעם אחת ללא צורך בהגדרת ערכי המשתנים באופן מלאכותי נשתמש במשפט **do/while**.

משפט do/while מקבל את הצורה הבאה:

**do**
embedded-statement
**while** ( Boolean-expression )

מאחר וחישוב ביטוי התנאי Boolean-expression של המשפט while קורה לאחר embedded-statement, מובטח לך ש-embedded-statement יבוצע פעם אחת לפחות. כאן כתבתי מחדש את דוגמת יישום המספרים תוך שימוש במשפט **do/while**:

```
using System;

class DoWhileApp
{
   const int MIN = 1;
   const int MAX = 10;
   const string QUIT_CHAR = "Q";

   public static void Main()
   {
      Random rnd = new Random();
      double correctNumber;

      string inputString;
      int userGuess = -1;

      bool userHasNotQuit = true;

      do
      {
         correctNumber = rnd.NextDouble() * MAX;
         correctNumber = Math.Round(correctNumber);

         Console.Write
            ("Guess a number between {0} and {1}...({2} to quit)",
            MIN, MAX, QUIT_CHAR);
         inputString = Console.ReadLine();

         if (0 == string.Compare(inputString, QUIT_CHAR, true))
            userHasNotQuit = false;
         else
         {
            userGuess = inputString.ToInt32();
            Console.WriteLine
               ("The correct number was {0}\n", correctNumber);
         }
      } while (userGuess != correctNumber && userHasNotQuit);

      if (userHasNotQuit && userGuess == correctNumber)
      {
         Console.WriteLine("Congratulations!");
      }
      else // wrong answer!
      {
         Console.WriteLine("Maybe next time!");
      }
   }
}
```

יישום זה מתפקד כמו היישום עם המשפט while. ההבדל היחיד הוא איך מתבצעת בקרת הלולאה. בפועל, השימוש במשפט while שכיח יותר מאשר במשפט do/while. עם זאת, היות וניתן לשלוט בכניסה ללולאה ביתר קלות עם אתחול של משתנה בוליאני, העדפת אחד המשפטים על פני האחר היא אישית.

## משפט for

משפט האיטרציה שבו תיתקל ותשתמש בדרך כלל הוא משפט **for**. משפט זה מורכב משלושה מקטעים. מקטע אחד משמש לאתחול הלולאה בראשיתה והוא מתבצע פעם אחת בלבד. המקטע השני הוא מבחן תנאי הקובע אם יש לבצע שוב את הלולאה. המקטע האחרון נקרא מקטע האיטרציה ומשמש בדרך כלל (אך לא בהכרח) לקידום מונה בקרת המשך הלולאות. בדרך כלל נבדק מונה זה במקטע השני (מקטע התנאי). משפט for הוא בעל הצורה הבאה:

```
for (initialization; Boolean-expression; step) {
        embedded-statement
}
```

שים לב שכל אחד ממקטעים אלה (initialization, Boolean-expression, step) הוא אופציונלי. כאשר תוצאת Boolean-expression היא false, הבקרה עוברת מראש הלולאה לשורה הבאה לאחר embedded-statement. לפיכך, המשפט for פועל בדיוק כמו משפט while בתוספת שני מקטעים (אתחול וקידום). הנה דוגמה של משפט for המציגה את כל תווי ASCII הניתנים להדפסה:

```
using System;

class ForTestApp
{
   const int StartChar = 33;
   const int EndChar = 125;

   static public void Main()
   {
      for (int i = StartChar; i <= EndChar; i++)
      {
         Console.WriteLine("{0}={1}", i, (char)i);
      }
   }
}
```

סדר האירועים עבור לולאת for זו הוא כדלקמן:

1. הקצאת משתנה מסוג ערך (i) במחסנית ואתחולו עם הערך 33. שים לב שמשתנה זה יוצא מטווח ההכרה ברגע שהלולאה מסתיימת.

2. המשפט בגוף הלולאה יתבצע כל זמן שערכו של המשתנה i קטן מ- 126. בדוגמה זו השתמשתי במשפט מורכב. עם זאת, מאחר ולולאת for זו מכילה משפט אחד בלבד, ניתן לכתוב משפט זה גם ללא הסוגריים המסולסלים ולקבל את אותן התוצאות.

3. לאחר כל איטרציה, כלומר ביצוע הלולאה, יגדל ערכו של המשתנה i באחד.

## לולאות מקוננות

החלק embedded-statement של לולאת for יכול להכיל לולאות for אחרות, אשר בדרך כלל מתייחסים אליהן כאל **לולאות מקוננות** (nested loops). בדוגמה בסעיף קודם הוספתי לולאה מקוננת הגורמת ליישום להדפיס שלושה תווים במקום אחד בכל שורה :

```
using System;

class NestedForApp
{
   const int StartChar = 33;
   const int EndChar = 125;
   const int CharactersPerLine = 3;

   static public void Main()
   {
      for (int i = StartChar; i <= EndChar; i+=CharactersPerLine)
      {
         for (int j = 0; j < CharactersPerLine;  j++)
         {
            Console.Write("{0}={1} ", i+j, (char)(i+j));
         }
         Console.WriteLine("");
      }
   }
}
```

שים לב שהמשתנה i המוגדר בלולאה החיצונית נשאר זמין עבור הלולאה הפנימית. עם זאת, המשתנה j אינו זמין ללולאה החיצונית.

## השימוש באופרטור הפסיק

הפסיק (,) יכול לשמש לא רק כתו הפרדה ברשימת ארגומנטים של שיטה, אלא גם כאופרטור במשפט for. ניתן להשתמש באופרטור הפסיק בחלק האתחול וגם בחלק האיטרציה של משפט for לתחימת משפטים מרובים המעובדים באופן סדרתי. בדוגמה הבאה השתמשתי בלולאה המקוננת הקודמת והמרתי אותה ללולאת for יחידה באמצעות שימוש באופרטור הפסיק :

```
using System;

class CommaOpApp
{
   const int StartChar = 33;
   const int EndChar = 125;

   const int CharactersPerLine = 3;

   static public void Main()
   {
      for (int i = StartChar, j = 1; i <= EndChar; i++, j++)
      {
         Console.Write("{0}={1} ", i, (char)i);
         if (0 == (j % CharactersPerLine))
         {
            // New line if j is divisible by 3.
            Console.WriteLine("");
         }
      }
   }
}
```

השימוש באופרטור הפסיק במשפט for עשוי להיות רב-עוצמה, אך עלול גם להוביל לכתיבת קוד מכוער וקשה לתחזוקה. בדוגמה הבאה, למרות תוספת של קבועים, השימוש באופרטור הפסיק הוא נכון מבחינה טכנית, אך נעשה באופן בלתי-ראוי:

```
using System;

class CommaOp2App
{
   const int StartChar = 33;
   const int EndChar = 125;

   const int CharsPerLine = 3;
   const int NewLine = 13;
   const int Space = 32;

   static public void Main()
   {
      for (int i = StartChar, extra = Space;
         i <= EndChar;
         i++, extra = ((0 == (i - (StartChar-1)) % CharsPerLine)
         ? NewLine : Space))
      {
         Console.Write("{0}={1} {2}", i, (char)i, (char)extra);
      }
   }
}
```

# משפט foreach

מזה זמן רב, לשפות כמו Visual Basic יש משפט שתוכנן במיוחד לאיטרציה על מערכים ואוספים. גם לשפת #C יש מבנה כזה, משפט **foreach**, המקבל את הצורה הבאה:

```
foreach (type in expression) {
embedded-statement
}
```

הסתכל על מחלקת המערך הבאה:

```
class MyArray
{
   public ArrayList words;

   public MyArray()
   {
      words = new ArrayList();
      words.Add("foo");
      words.Add("bar");
      words.Add("baz");
   }
}
```

על סמך היכרותך עם משפטי האיטרציה השונים עד כה, אתה יודע שניתן לגשת לאלמנטים של מערך זה באמצעות מספר משפטים שונים. עם זאת, עבור רוב מתכנתי Java ו- ++C, הדרך ההגיונית ביותר לכתוב את היישום תהיה בדומה לזו:

```
using System;
using System.Collections;

class MyArray
{
   public ArrayList words;

   public MyArray()
   {
      words = new ArrayList();
      words.Add("foo");
      words.Add("bar");
      words.Add("baz");
   }
}

class Foreach1App
{
   public static void Main()
```

```
{
      MyArray myArray = new MyArray();

      for (int i = 0; i < myArray.words.Count; i++)
      {
         Console.WriteLine("{0}", myArray.words[i]);
      }
   }
}
```

אולם, גישה זו מולידה מספר בעיות פוטנציאליות:

- אם משתנה האתחול i של המשפט for מאותחל באופן שגוי, לא ניתן יהיה לבצע איטרציה על כל הרשימה.
- אם ביטוי התנאי של המשפט for הוא שגוי, לא ניתן יהיה לבצע איטרציה על כל הרשימה.
- אם קידום המונה של המשפט for אינו נכון, לא ניתן יהיה לבצע איטרציה על כל הרשימה.
- לאוספים ולמערכים יש שיטות ומאפיינים שונים לקבלת מספר האלמנטים שלהם.
- לאוספים ולמערכים יש סמנטיקות שונות לחילוץ אלמנט מסוים.
- המשפט בגוף לולאת for צריך לחלץ את האלמנט לתוך משתנה מהסוג הנכון.

ובכן, שגיאת קוד עלולה לקרות באופנים רבים. באמצעות משפט foreach אפשר להימנע מבעיות אלו ולבצע איטרציה על כל אוסף או מערך באופן אחיד. כך נכתוב מחדש את הקוד הקודם עם שימוש במשפט foreach:

```
using System;
using System.Collections;

class MyArray
{
   public ArrayList words;

   public MyArray()
   {
      words = new ArrayList();
      words.Add("foo");
      words.Add("bar");
      words.Add("baz");
   }
}
```

```
class Foreach2App
{
   public static void Main()
   {
      MyArray myArray = new MyArray();

      foreach (string word in myArray.words)
      {
         Console.WriteLine("{0}", word);
      }
   }
}
```

שים לב שהשימוש במשפט foreach הוא הרבה יותר אינטואיטיבי. מובטח לך שתקבל כל אלמנט מאחר ואינך צריך להגדיר את הלולאה ולקבל את מספר האלמנטים באופן ידני, והמשפט מציב אוטומטית את האלמנט במשתנה משלך. כל שאתה נדרש הוא רק להתייחס למשתנה זה במשפט גוף הלולאה.

# הסתעפות עם משפטי דילוג

בתוך המשפטים הנמצאים בגוף כל אחד ממשפטי האיטרציה שנסקרו בסעיפים הקודמים, ניתן לשלוט בזרימת הביצוע עם אחד או יותר משפטים הנקראים באופן קולקטיבי משפטי דילוג: break, continue, goto ו-return.

## משפט break

משתמשים במשפט **break** בכדי לסיים את הלולאה הסוגרת הנוכחית, או כדי לסיים את משפט התנאי שבו הוא מופיע. לאחר מכן עוברת הבקרה לשורה הבאה לאחר אותה לולאה, או לאחר המשפט הכלול במשפט התנאי. משפט break הוא בעל התחביר הפשוט ביותר מכל המשפטים ואינו כולל סוגריים או ארגומנטים. הוא מקבל את הצורה הבאה בנקודה שבה מעוניינים לצאת מלולאה או משפט תנאי:

**break**

בדוגמה הבאה, היישום ידפיס כל מספר מ- 1 עד 100 המתחלק ב- 6. עם זאת, כאשר המונה מגיע ל- 66, המשפט break יגרום לסיום הלולאה for.

```
using System;

class BreakTestApp
{
   public static void Main()
```

```
    {
        for (int i = 1; i <= 100; i ++)
        {
            if (0 == i % 6)
            {
                Console.WriteLine(i);
            }

            if (i == 66)
            {
                break;
            }
        }
    }
}
```

## יציאה מלולאות אינסופיות

שימוש נוסף במשפט break הוא ליצור לולאה אינסופית שבה הבקרה מועברת אל מחוץ ללולאה רק כאשר מגיעים למשפט break. הדוגמה הבאה ממחישה שיטה לכתיבת הדוגמה של ניחוש המספרים (שהוצגה מקודם) עם משפט break המשמש ליציאה מן הלולאה ברגע שהמשתמש מקליד את האות Q. שים לב ששיניתי את המשפט while ל-while(true) כך שהלולאה לא תסתיים עד שנתקלים במשפט break.

```
using System;

class InfiniteLoopApp
{
    const int MIN = 1;
    const int MAX = 10;
    const string QUIT_CHAR = "Q";

    public static void Main()
    {
        Random rnd = new Random();
        double correctNumber;

        string inputString;
        int userGuess;

        bool correctGuess = false;
        bool userQuit = false;
```

```
        while(true)
        {
            correctNumber = rnd.NextDouble() * MAX;
            correctNumber = Math.Round(correctNumber);

            Console.Write
                ("Guess a number between {0} and {1}...({2} to quit)",
                MIN, MAX, QUIT_CHAR);
            inputString = Console.ReadLine();

            if (0 == string.Compare(inputString, QUIT_CHAR, true))
            {
                userQuit = true;
                break;
            }
            else
            {
                userGuess = inputString.ToInt32();
                correctGuess = (userGuess == correctNumber);

                if (correctGuess)
                {
                    break;
                }
                else
                {

                    Console.WriteLine
                        ("The correct number was {0}\n", correctNumber);
                }
            }
        }

        if (correctGuess && !userQuit)
        {
            Console.WriteLine("Congratulations!");
        }
        else
        {
            Console.WriteLine("Maybe next time!");
        }
    }
}
```

כאן יכולתי להשתמש בלולאת for ריקה מהצורה for(;;) במקום while(true). שתי הלולאות פועלות באופן זהה ולכן הבחירה היא רק על פי העדפה אישית.

## משפט continue

כמו משפט break, משפט **continue** מאפשר לשנות את ביצוע הלולאה. עם זאת, במקום לסיים את המשפט הטבוע בלולאה הנוכחית, משפט continue מפסיק את האיטרציה הנוכחית ומחזיר את הבקרה לראש הלולאה עבור האיטרציה הבאה. בדוגמה הבאה יש מערך מחרוזות שבו אני רוצה לבדוק כפילויות. דרך אחת לבצע זאת היא איטרציה על המערך עם לולאה מקוננת והשוואה בין כל שני אלמנטים. יחד עם זאת, איני רוצה להשוות בין אלמנט לעצמו אחרת אקבל מספר לא נכון של כפילויות. לפיכך, אם אינדקס i לתוך המערך זהה לאינדקס j לתוך המערך, הרי שמדובר באותו אלמנט ולכן לא אשווה בין השניים. במקרה זה אני משתמש במשפט continue בכדי להפסיק את ביצוע האיטרציה הנוכחית ולהעביר את הבקרה חזרה לראש הלולאה.

```
using System;
using System.Collections;

class MyArray
{
   public ArrayList words;

   public MyArray()
   {
      words = new ArrayList();
      words.Add("foo");
      words.Add("bar");
      words.Add("baz");
      words.Add("bar");
      words.Add("ba");
      words.Add("foo");
   }
}

class ContinueApp
{
   public static void Main()
   {
      MyArray myArray = new MyArray();
      ArrayList dupes = new ArrayList();

      Console.WriteLine("Processing array...");
      for (int i = 0; i < myArray.words.Count; i++)
      {
         for (int j = 0; j < myArray.words.Count; j++)
```

```
            {
                if (i == j) continue;

                if (myArray.words[i] == myArray.words[j] && !dupes.Contains(j))
                {
                    dupes.Add(i);
                    Console.WriteLine("'{0}' appears on lines {1} and {2}",
                        myArray.words[i],
                        i + 1,
                        j + 1);
                }
            }
        }
        Console.WriteLine("There were {0} duplicates found",
            ((dupes.Count > 0) ? dupes.Count.ToString() : "no"));
    }
}
```

בדוגמה זו יכולתי להשתמש בלולאת foreach בכדי לבצע איטרציה על המערך. עם זאת, במקרה מסוים זה רציתי לדעת לאיזה אלמנט ניגשתי בכל פעם, כך שהדרך הטובה ביותר היתה להשתמש בלולאת for.

# משפט goto

בהיסטוריה של התכנות אין מרכיב מושמץ יותר מאשר משפט **goto**. לכן, לפני שניגש לתחביר ולמספר שימושים של משפט goto, הבה ננסה להבין מדוע ישנם אנשים עם תחושות חזקות שלא להשתמש במשפט זה ואת סוגי הבעיות שניתן לפתור באמצעותו.

## מעט היסטוריה על משפט goto

ההסתייגות ממשפט goto גובשה במאמר שיצא לאור בשנת 1968 בשם "Go To Statement Considered Harmful" מאת Edsger W. Dijkstra. באותה נקודת זמן היסטורית של התכנות היה ויכוח על הסוגיה של תכנות מובנה. לרוע המזל, פחות תשומת לב הוקדשה לסוגיות הכוללות הקשורות בתכנות מובנה מאשר בפרט קטן יחסית: האם משפטים מסוימים, כגון go to (כיום בדרך כלל מילת המפתח goto), צריכים להיות נוכחים בשפות תכנות מודרניות. התברר שאנשים רבים קיבלו את עצתו של Dijkstra והסיקו מסקנה קיצונית, שהשימוש במשפט goto לצורותיו הוא רע ויש להימנע ממנו בו בכל מחיר.

הבעיה ב-goto אינה מילת המפתח עצמה אלא דווקא השימוש בה במקומות שאינם ראויים. משפט goto יכול להיות כלי שימושי בבנייה של זרימת תוכנית וניתן לכתוב באמצעותו קוד מובן יותר מזה המתקבל ממנגנונים אחרים של איטרציה והסתעפות. דוגמה אחת לכך היא בעיית "לולאה וחצי" ("loop-and-a-half"), כפי שכינה אותה Dijkstra. הנה תיאור הזרימה המסורתי של בעיה זו בפסידו-קוד:

```
loop
   read in a value
   if value == sentinel then exit
   process the value
end loop
```

היציאה מהלולאה מושגת רק על ידי ביצוע המשפט exit במרכז הלולאה. עם זאת, המעגל של loop/exit/end loop יכול להטריד מספר אנשים, אשר על פי תפיסתם כל השימושים ב-goto הם רעים. אנשים אלה היו כותבים קוד כמו זה:

```
read in a value
while value != sentinel
   process the value
   read in a value
end while
```

למרבה הצער, כפי שציין Eric S. Roberts מאוניברסיטת Stanford, הגישה השנייה לוקה בשני חסרונות עיקריים. ראשית, היא מחייבת כפילות במשפט(ים) הנדרש לקריאת ערך. שכפול קוד יוצר בעיית תחזוקה ברורה כי כל שינוי במשפט אחד מחויב גם באחר. הבעיה השנייה אולי גם מזיקה יותר. המפתח לכתיבת קוד מוצק שקל להבינו ולכן גם לתחזקו, הוא לכתוב קוד שניתן להבינו על פי דרך התפיסה הטבעית. אם רוצים לתאר מה שקוד זה מנסה לעשות בצורה אינטואיטיבית, הפתרון היה מתואר כך: תחילה, אני צריך לקרוא ערך. אם ערך זה הוא ציון כלשהו (sentinel), אני מפסיק. אם לא, אני מעבד את הערך וממשיך לערך הבא. לפיכך, דווקא הקוד שמשמיט את משפט exit פועל למעשה נגד האינטואיציה מאחר וגישה זו הופכת את דרך החשיבה הטבעית על הבעיה. כעת נסתכל על מספר מצבים בהם משפט goto יכול להיות הפתרון הטוב ביותר לבניית בקרת זרימה.

## השימוש במשפט goto

משפט goto יכול לקבל את אחת מהצורות הבאות:

**goto** identifier
**goto case** constant-expression
**goto default**

בשימוש הראשון של משפט goto, היעד של ה-identifier (מזהה) הוא משפט תווית. משפט תווית הוא מהצורה הבאה:

identifer:

אם התווית אינה קיימת בשיטה הנוכחית, נוצרת שגיאת זמן-הידור. כלל חשוב אחר שיש לזכור הוא שמשפט goto יכול לשמש לדילוג החוצה מלולאה מקוננת. עם זאת, אם המשפט goto אינו נמצא בגבולות טווח ההכרה של התווית, נוצרת שגיאת זמן-הידור. לכן, אינך יכול לדלג לתוך לולאה מקוננת.

בדוגמה הבאה מתבצעת איטרציה על מערך פשוט וקוראים כל ערך עד שנתקלים בציון כלשהו (sentinel) ומתרחשת יציאה מהלולאה. שים לב שהמשפט goto פועל ממש כמו משפט break בכך שהוא גורם לזרימת התוכנית לדלג אל מחוץ ללולאת foreach.

```
using System;
using System.Collections;

class MyArray
{
   public ArrayList words;
   public const string TerminatingWord = "stop";

   public MyArray()
   {
      words = new ArrayList();

      for (int i = 1; i <= 5; i++) words.Add(i.ToString());
      words.Add(TerminatingWord);
      for (int i = 6; i <= 10; i++) words.Add(i.ToString());
   }
}

class Goto1App
{
   public static void Main()
   {
      MyArray myArray = new MyArray();

      Console.WriteLine("Processing array...");

      foreach (string word in myArray.words)
      {
         if (word == MyArray.TerminatingWord) goto finished;
         Console.WriteLine(word);
      }

      finished:
         Console.WriteLine("Finished processing array");
   }
}
```

מישהו יכול לטעון שניתן היה להשתמש במשפט break בתוכנית זו באותה אפקטיביות וללא צורך בתווית. אנו נבחן צורות אחרות של משפט goto ונראה שיש בעיות הניתנות לפתרון רק עם משפט זה.

בסעיף העוסק במשפט switch דנתי בעובדה ש-#C אינה תומכת בגלישות. גם אם גלישות היו נתמכות, זה לא היה פותר את הבעיה הבאה. נחזור שוב למחלקה Payment (בה השתמשנו מקודם) המקבלת מספר צורות תשלום: Visa, American Express, Master Card, cash (מזומן) ו-charge off (זיכוי - זהו אשראי בסיסי). מאחר ו-Visa, American Express ו- Master Card הם כולם כרטיסי אשראי, נוכל לצרף אותם לתווית case בודדת ולעבד את כולם באותו אופן. במקרה של זיכוי (charge off), עלינו לקרוא לשיטות ייחודיות למקרה זה, ובמקרה של קנייה במזומן עלינו רק להדפיס קבלה. בנוסף לכך, היינו רוצים להדפיס קבלה עבור כל אחד מהמקרים. איך נוכל להגדיר שלושה מקרים נפרדים כך ששני המקרים הראשונים - כרטיס אשראי וזיכוי - יסתעפו לתווית של קנייה במזומן לאחר סיום עיבודם? כפי שתוכל לראות בקוד הבא, בעיה זו היא דוגמה טובה מתי צריך להשתמש במשפט goto:

```
using System;

enum Tenders : int
{
   ChargeOff,
   Cash,
   Visa,
   MasterCard,
   AmericanExpress
};

class Payment
{
   public Payment(Tenders tender)
   {
      this.Tender = tender;
   }

   protected Tenders tender;
   public Tenders Tender
   {
      get
      {
         return this.tender;
      }
```

```
    set
    {
        this.tender = value;
    }
}

protected void ChargeOff()
{
    Console.WriteLine("Charge off.");
}

protected bool ValidateCreditCard()
{
    Console.WriteLine("Card approved.");
    return true;
}

protected void ChargeCreditCard()
{
    Console.WriteLine("Credit Card charged");
}

protected void PrintReceipt()
{
    Console.WriteLine("Thank you and come again.");
}

public void ProcessPayment()
{
    switch ((int)(this.tender))
    {
        case (int)Tenders.ChargeOff:
            ChargeOff();
            goto case Tenders.Cash;

        case (int)Tenders.Visa:
        case (int)Tenders.MasterCard:
        case (int)Tenders.AmericanExpress:
            if (ValidateCreditCard())
                ChargeCreditCard();
            goto case Tenders.Cash;

        case (int)Tenders.Cash:
            PrintReceipt();
            break;
```

```
        default:
            Console.WriteLine("\nSorry - Invalid tender.");
            break;
        }
    }
}

class GotoCaseApp
{
    public static void Main()
    {
        Payment payment = new Payment(Tenders.Visa);
        payment.ProcessPayment();
    }
}
```

במקום לפתור את הבעיה בצורה מנוגדת לחשיבה טבעית, אנו פשוט אומרים למהדר שכאשר מסתיים תהליך העיבוד של כרטיס האשראי או הזיכוי, אנו רוצים להסתעף לתווית -cash. כאן יש לציין שאם מסתעפים החוצה מתווית case ב-C#, אין להשתמש במשפט break, דבר שגורם לשגיאת הידור מסוג "קוד שלא ניתן להגיע אליו".

הצורה האחרונה של משפט goto מאפשרת להסתעף לתווית default במשפט switch, ובכך מספקת אמצעי אחר לכתיבת גוש קוד יחיד שניתן לביצוע כתוצאה מחישובים מרובים של משפט switch.

## משפט return

למשפט **return** יש שני תפקידים. הוא מגדיר ערך שיש להחזיר לשיטה הקוראת לקוד המתבצע כרגע (כאשר הקוד הנוכחי אינו מוגדר כמחזיר void), והוא גורם לחזרה מיידית לשיטה הקוראת. משפט return מוגדר עם התחביר הבא:

**return** [return-expression];

כאשר המהדר נתקל במשפט return בתוך שיטה ומשפט זה מגדיר return-expression, הוא מחשב האם ניתן להמיר ערך זה לצורה שתואמת את הערך שהשיטה צריכה להחזיר כמוגדר בכותר שלה. תוצאת ההמרה הזו היא זו אשר מוחזרת לקוד הקורא.

כאשר משתמשים במשפט return עם טיפול בחריגים (ראה בפרק 12 טיפול בשגיאות באמצעות חריגים), יש לוודא שמבינים מספר חוקים. אם המשפט return הוא בלוק try המכיל בלוק finally משויך, הבקרה מועברת בפועל לשורה הראשונה של הבלוק finally, וכאשר מסתיים הביצוע של בלוק קוד זה, הבקרה מועברת חזרה לקורא. אם הבלוק try מקונן בתוך בלוק try אחר, הבקרה מועברת אחורה במעלה השרשרת באותו האופן עד לסיום הביצוע של בלוק finally האחרון.

# סיכום

משפטי התנאי ב-#C מאפשרים בקרה על זרימת התוכנית. שלוש הקטגוריות השונות של משפטי זרימה כוללות משפטי בחירה, כגון if ו-switch, משפטי איטרציה (for ,while ו-foreach) ומשפטי דילוג שונים (break, continue, goto ו-return). ההחלטה מהם המשפטים הטובים ביותר שיש להשתמש בהם על סמך החומר שהוצג בפרק זה תעזור לך לכתוב יישומים מובנים טובים יותר וקלים יותר לתחזוקה.

# פרק 12

# טיפול בשגיאות באמצעות חריגים

אחד היעדים העיקריים של CLR הוא להימנע משגיאות זמן-ריצה (באמצעות תכונות כמו זיכרון אוטומטי וניהול משאבים כאשר משתמשים בקוד מנוהל) או לפחות לתפוס אותן בזמן ההידור (באמצעות CTS). עם זאת, שגיאות מסוימות ניתן לתפוס רק בזמן ריצה, ולכן חייבים להשתמש באמצעי עקבי לטיפול בשגיאות עבור כל השפות המצייתות לחוקים של CLS. פרק זה מתמקד במערכת הטיפול בשגיאות המיושמת על ידי CLR - טיפול בחריגים (exceptions).

בפרק זה תלמד תחילה על המכניזם הכללי והתחביר הבסיסי של טיפול בחריגים. לאחר שתרכוש ידע בסיסי זה, נשווה בין טיפול בחריגים לבין שיטות שכיחות יותר כיום לטיפול בשגיאות ותגלה את היתרונות שיש לטיפול בחריגים על פני טכניקות אחרות אלו. לאחר מכן נדון במספר סוגיות מסוימות של טיפול בחריגים במסגרת NET, כמו השימוש במחלקה Exception וגזירת מחלקות Exception מותאמות. החלק האחרון של הפרק יעסוק בסוגיית תכנון נכון של מערכת המיישמת טיפול בחריגים.

## סקירה כללית על טיפול בחריגים

**חריגים** (exceptions) הם מצבי שגיאות שמתעוררים כאשר הזרימה הרגילה של נתיב הקוד - כלומר, סדרת הקריאות לשיטות במחסנית הקריאות - היא נמהרת או שאינה מעשית. כאן חיוני להבין את השוני בין חריג לבין אירוע צפוי (כמו הגעה לסוף קובץ). אם שיטה קוראת קובץ באופן סדרתי, ידוע שבנקודה כלשהי היא תגיע לסוף הקובץ.

לכן אירוע זה כמעט שאינו נחשב **חריג** מטבעו וכמובן שאינו מונע מהיישום להתקדם. לעומת זאת, אם מנסים לקרוא מתוך קובץ ומערכת ההפעלה מודיעה על זיהוי שגיאת דיסק, זהו כמובן מצב חריג אשר בוודאות משפיע על הזרימה הנורמלית של השיטה המנסה להמשיך בקריאת הקובץ.

רוב החריגים כרוכים בבעיה נוספת: הקשר (context). לדוגמה, נניח שאתה כותב קוד בעל צמידות חזקה (cohesive code), שבו שיטה אחת אחראית לפעולה אחת. הקוד שלך עשוי להיראות בדומה לזה:

```
public void Foo()
{
   File file = OpenFile(String fileName);
   while (!file.IsEOF())
   {
      String record = file.ReadRecord();
   }
CloseFile();
}
public void OpenFile(String fileName)
{
   // Attempt to lock and open file.
}
```

שים לב שאם השיטה OpenFile נכשלת, היא אינה מסוגלת לטפל בשגיאה. זאת מכיון שהיא אחראית אך ורק לפתיחת קבצים. היא אינה יכולה לקבוע אם אי-היכולת לפתוח את הקובץ המסוים היא שגיאה נוראית או מטרד שולי. לפיכך, אנו אומרים שהשיטה OpenFile אינה יכולה להתמודד עם השגיאה מכיון שאינה נמצאת בהקשר הנכון.

זו כל הסיבה לקיום של טיפול בחריגים: שיטה אחת מזהה מצב חריג ובמקרה אינה נמצאת בהקשר הנכון בכדי להתמודד עם השגיאה. היא מאותתת לזמן הריצה ששגיאה התרחשה. זמן הריצה מתגלגל אחורנית במעלה מחסנית הקריאות עד שנמצאת שיטה המסוגלת לטפל כיאות במצב השגיאה. ללא ספק, החשיבות לכך גדלה אף יותר אם נתיב הקוד הוא בעומק של חמש שיטות כאשר השיטה החמישית מגיעה למצב שגיאה ורק השיטה הראשונה יכולה לטפל בשגיאה בצורה נכונה. כעת נלמד את התחביר המשמש לטיפול בחריגים.

# תחביר בסיסי של טיפול בחריגים

הטיפול בחריגים מורכב מארבע מילות מפתח בלבד: try, catch, throw ו-finally. אופן הפעולה של מילים אלו הוא פשוט ומובן מאליו. כאשר שיטה נכשלת במשימתה ואינה יכולה להתקדם - כלומר, כאשר היא מזהה מצב חריג - היא זורקת חריג לשיטה הקוראת באמצעות מילת המפתח throw. השיטה הקוראת, בהנחה שיש לה הקשר מספיק בכדי לטפל בחריג, מקבלת את החריג באמצעות מילת המפתח catch ומחליטה באיזו פעולה יש לנקוט. בסעיפים הבאים נבחן את הסמנטיקה המנהלת את אופן הזריקה והתפיסה של חריגים וגם קטעי קוד הממחישים זאת.

## Throwing an Exception

כאשר שיטה צריכה להודיע לשיטה הקוראת שהתרחשה שגיאה, היא משתמשת במילת המפתח **throw** (זרוק) באופן הבא:

**throw** statement:
**throw** $\text{expression}_{\text{opt}}$;

יותר מאוחר נדון בדרכים השונות בהם תוכל לזרוק חריגים (throwing an exception). לעת עתה, מספיק להבין שכאשר זורקים חריג, נדרשים לזרוק אובייקט מסוג System.Exception (או מחלקה נגזרת). להלן דוגמה של שיטה הקובעת שקרתה שגיאה שלא ניתן להתמודד איתה והיא צריכה לזרוק חריג לשיטה הקוראת. שים לב איך השיטה יוצרת מופע אובייקט System.Exception חדש ואז זורקת אובייקט חדש זה כלפי מעלה למחסנית הקריאות.

```
public void SomeMethod()
{
// Some error determined.
throw new Exception();
}
```

## Catching an Exception

מכיון ששיטה יכולה לזרוק חריג, ללא ספק חייב להיות חלק הדדי במשוואה זו שבו משהו צריך לתפוס את החריג הנזרק. מילת המפתח **catch** מגדירה בלוק קוד שיש לעבד אותו כאשר נתפס חריג מסוג נתון. הקוד הכלול בבלוק זה נקרא **מטפל בחריג** (exception handler).

יש לזכור שלא כל שיטה חייבת להתמודד עם כל חריג אפשרי שנזרק, במיוחד משום שלשיטה אין אולי את ההקשר המספיק הנדרש לטפל במידע השגיאה. אחרי הכל, העיקרון בטיפול בחריגים הוא ששגיאות צריכות להיות מטופלות באמצעות קוד שמכיל מידה מספקת של הקשר בכדי לעשות זאת בצורה נכונה (ראה את הסעיף "טיפול בשגיאות בהקשר הנכון" בהמשך הפרק). לעת עתה נעסוק במצבים בהם שיטה עומדת לתפוס כל חריג שנזרק מהשיטה הנקראת. משתמשים בשתי מילות מפתח לתפוס חריג: **try** (נסה) ו-**catch** (תפוס).

בכדי לתפוס חריג, יש לתחום את הקוד שאותו מנסים לבצע בבלוק try ואז להגדיר בבלוק catch באילו סוגים של חריגים קוד זה מסוגל לטפל. כל המשפטים בבלוק try מעובדים בסדר הרגיל אלא אם נזרק חריג על ידי אחת מהשיטות הנקראות. אם זה קורה, הבקרה מועברת לשורה הראשונה של בלוק catch המתאים. ב"בלוק catch המתאים" כוונתי לבלוק המוגדר כך שיתפוס את סוג החריג שנזרק. הנה דוגמה של שיטה (Foo) הקוראת ותופסת חריג שנזרק על ידי שיטה אחרת (Bar):

```
public void Foo()
{
   try
   {
      Bar();
   }
   catch(System.Exception e)
   {
      // Do something with error condition.
   }
}
```

נשאלת השאלה, מה קורה אם השיטה Bar זורקת חריג שהשיטה Foo אינה תופסת? זהו גם המקרה שבו הקריאה ל-Bar לא נמצאת בבלוק try. התוצאה תלויה בתכנון של היישום. כאשר נזרק חריג, הבקרה מועברת אחורנית במעלה מחסנית הקריאות עד שנמצאת שיטה המכילה בלוק catch עבור אותו סוג של חריג. אם לא נמצאת שיטה עם בלוק catch מתאים, ביצוע היישום מופסק. לפיכך, אם שיטה קוראת לשיטה אחרת שזורקת חריג, תכנון המערכת צריך להיות כזה שבו חייבת להיות שיטה כלשהי במחסנית הקריאות שמסוגלת לטפל בחריג.

## Rethrowing an Exception

יש זמנים בהם שיטה תופסת חריג, עושה כל שביכולתה במסגרת ההקשר שלה ואז מגלגלת אותו הלאה, כלומר זורקת את החריג שוב (rethrows) חזרה כלפי מעלה למחסנית הקריאות. הדבר נעשה בפשטות באמצעות מילת המפתח **throw**.

הנה דוגמה:

```
using System;

class RethrowApp
{
   static public void Main()
   {
      Rethrow rethrow = new Rethrow();

      try
      {
         rethrow.Foo();
      }
      catch(Exception e)
      {
         Console.WriteLine(e.Message);
      }
   }

   public void Foo()
   {
      try
      {
         Bar();
      }
      catch(Exception)
      {
         // Handle error.
         throw;
      }
   }

   public void Bar()
   {
      throw new Exception("thrown by Rethrow.Bar");
   }
}
```

בדוגמה זו, Main קוראת ל-Foo, אשר קוראת ל-Bar. Bar זורקת חריג ש-Foo תופסת. בנקודה זו Foo מבצעת עיבוד מסוים ואז זורקת שוב את החריג אחורנית למחסנית הקריאות עד לשיטה Main (תחילת התוכנית) באמצעות מילת המפתח throw.

## ניקוי בעזרת finally

קיימת סוגיה בלתי נפרדת מטיפול בחריגים והיא ההבטחה שקטע קוד מסוים יתבצע תמיד ללא קשר אם החריג נתפס או לא. לדוגמה, נניח שאתה מקצה משאבים כגון התקן פיסי או קובץ נתונים. לאחר מכן נניח שאתה פותח את אותו המשאב וקורא לשיטה שזורקת חריג. ללא תלות אם השיטה יכולה להמשיך לעבוד עם המשאב או לא, עדיין אתה חייב לשחרר את המשאב שהוקצה או לסגור את הקובץ. כאן משתמשים במילת המפתח **finally** (סופי), בדומה לזה:

```
using System;

public class ThrowException1App
{
   public static void ThrowException()
   {
      throw new Exception();
   }

   public static void Main()
   {
      try
      {
         Console.WriteLine("try...");
      }
      catch(Exception e)
      {
         Console.WriteLine("catch...");
      }
      finally
      {
         Console.WriteLine("finally");
      }
   }
}
```

כפי שאתה רואה, קיומה של מילת המפתח finally מונע ממך את הצורך להכניס קוד ניקוי גם בבלוק catch וגם לאחר הבלוקים try/catch. כעת, הקוד בבלוק finally יתבצע ללא תלות במה שקורה בתוך הבלוק try.

# השוואה בין טכניקות לטיפול בשגיאות

כעת, לאחר שהתוודענו לעקרונות של זריקת ותפיסת חריגים, הבה נקדיש מספר דקות בכדי להשוות בין גישות שונות לטיפול בשגיאות הננקטות בשפות תכנות.

הגישה הסטנדרטית לטיפול בשגיאות היתה בדרך כלל להחזיר קוד שגיאה לשיטה הקוראת. על שיטה זו מוטלת האחריות לפענח את הערך המוחזר ולפעול בהתאם. הערך המוחזר יכול להיות פשוט כמו סוג בסיסי של C או ++C או שיכול להיות מצביע לאובייקט חסון יותר המכיל את כל המידע הנדרש להעריך ולהבין לגמרי את השגיאה.

טכניקות מתוחכמות יותר לטיפול בשגיאות כוללות תת-מערכת שלמה עבור שגיאות שבה השיטה הנקראת מציינת את מצב השגיאה לתת-המערכת ואז מחזירה קוד שגיאה לשיטה הקוראת. זו קוראת לאחר מכן לפונקציה גלובלית אותה מייצאים מתת-מערכת השגיאות בכדי לקבוע את הסיבה לשגיאה האחרונה שנרשמה בתת-מערכת זו. תוכל למצוא דוגמה לגישה זו בערכת הפיתוח של Microsoft Open Database Connectivity (ODBC). מבלי להתחשב בסמנטיקה המדויקת, המושג הבסיסי נשאר אותו דבר: שיטה קוראת בדרך כלשהי לשיטה אחרת ובוחנת את הערך המוחזר בכדי לוודא את ההצלחה או הכישלון היחסיים של השיטה הנקראת. למרות שגישה זו היתה הסטנדרט במשך שנים רבות, היא פגומה קשות בכמה אופנים חשובים. הסעיף הבא מתאר מספר דרכים בהם לטיפול בחריגים יש יתרונות גדולים על שימוש בקודים מוחזרים.

## היתרון של טיפול בחריגים על פני קודים מוחזרים

כאשר משתמשים בקודים מוחזרים, השיטה הנקראת מחזירה קוד שגיאה ומצב השגיאה מטופל על ידי השיטה הקוראת. מאחר והטיפול בשגיאה מתרחש מחוץ לטווח ההכרה של השיטה הנקראת, אין כל ערובה לכך שהשיטה הקוראת תבדוק את קוד השגיאה המוחזר. כדוגמה, נניח שאתה כותב מחלקה בשם CommaDelimitedFile המאחדת יחד את הפונקציונליות של קריאה וכתיבה אל קבצים בהם הנתונים מופרדים על ידי פסיק. מחלקתך צריכה לחשוף בין השאר שיטות לפתיחת קובץ כזה ולקריאת נתונים ממנו. בשימוש בגישה הישנה של קוד מוחזר לדיווח על שגיאות, שיטות אלו היו מחזירות סוג משתנה מסוים שהשיטה הקוראת היתה צריכה לבדוק אותו בכדי לאמת את ההצלחה של הקריאה לשיטה. אם משתמש במחלקתך היה קורא לשיטה CommaDelimitedFile.Open ולאחר מכן מנסה לקרוא לשיטה CommaDelimitedFile.Read מבלי לבדוק אם הקריאה ל-Open הצליחה, זה היה גורם להרבה פחות מאשר התוצאה הרצויה, במיוחד מול הלקוח החשוב ביותר שלך שמולו אתה מדגים את היישום שכתבת. עם זאת, אם השיטה Open זורקת חריג, השיטה הקוראת נאלצת להתמודד עם העובדה שהשיטה Open נכשלה. זאת משום שבכל פעם ששיטה זורקת חריג, מתקיים תהליך שבו הבקרה מתגלגלת אחורה במעלה מחסנית הקריאות עד שהיא נתפסת. הנה דוגמה איך עשוי להיראות קוד זה:

```
using System;

class ThrowException2App
{
   class CommaDelimitedFile
   {
      protected string fileName;

      public void Open(string fileName)
      {
         this.fileName = fileName;
```

```
            // Attempt to open file
            // and throw exception upon error condition.
            throw new Exception("open failed");
        }

        public bool Read(string record)
        {
            // Code to read file.
            return false; // EOF
        }
    }

    public static void Main()
    {
        try
        {
            Console.WriteLine("attempting to open file");

            CommaDelimitedFile file = new CommaDelimitedFile();
            file.Open("c:\\test.csv");

            string record = "";

            Console.WriteLine("reading from file");

            while (file.Read(record) == true)
            {
                Console.WriteLine(record);
            }

            Console.WriteLine("finished reading file");
        }
        catch (Exception e)
        {
            Console.WriteLine(e.Message);
        }
    }
}
```

אם אחת מהשיטות, CommaDelimitedFile.Open או CommaDelimitedFile.Read, זורקת חריג, השיטה הקוראת נאלצת להתמודד איתו. אם השיטה הקוראת אינה תופסת את החריג ואף שיטה אחרת בנתיב הקוד הנוכחי לא מנסה לתפוס חריג מסוג זה, ביצוע היישום מופסק. שים לב במיוחד לעובדה שהצבת השיטה Open בבלוק מונעת קריאה פסולה לשיטה Read כאשר השיטה Open זורקת חריג. זאת מאחר ובקרה תכנותית עוברת מהשיטה Open בבלוק try לשורה הראשונה בבלוק catch. לפיכך, אחד מהיתרונות הגדולים ביותר של טיפול בחריגים על פני קודים מוחזרים הוא שיותר קשה להתעלם מחריגים מבחינה תכנותית.

## טיפול בשגיאות בהקשר הנכון

עיקרון כללי בתכנות טוב הוא הנוהג של כתיבת קוד בעל צמידות חזקה (tight cohesion), המתייחס למטרה או הייעוד של שיטה נתונה. שיטות המציגות צמידות חזקה הן כאלו המבצעות משימה בודדת. התועלת העיקרית בשימוש בצמידות חזקה בתכנות היא סיכוי גדול יותר לכך שהשיטה תהיה ניידת ותשמש בתסריטים שונים כאשר היא מבצעת פעולה אחת בלבד. שיטה כזו היא גם בוודאות קלה יותר לניפוי שגיאות ולתחזוקה. עם זאת, לקוד עם צמידות חזקה יש בעיה עיקרית ביחס לטיפול בשגיאות. הבה נבחן דוגמה הממחישה את הבעיה ואיך טיפול בחריגים פותר אותה.

בדוגמה זו, מחלקה (AccessDatabase) משמשת ליצירת ותפעול מסדי נתונים של Microsoft Access. נניח שמחלקה זו מכילה שיטה סטטית הנקראת GenerateDatabase. מאחר ושיטה זו תשמש ליצירת מסד נתונים חדש של Access, היא נדרשת לבצע מספר משימות לשם כך. לדוגמה, עליה ליצור את הקובץ הפיסי של מסד הנתונים, ליצור טבלאות מוגדרות (כולל שורות ועמודות הנדרשות על ידי היישום שלך) ולהגדיר את הסדרנים ואת היחסים הנדרשים. השיטה GenerateDatabase אולי צריכה גם ליצור מספר משתמשי ברירת מחדל והרשאות.

בעיית התכנון התכנותית היא כדלקמן: אם מתרחשת שגיאה בשיטה CreateIndexes, איזו שיטה תטפל בה וכיצד? ברור שבנקודה כלשהי השיטה שקראה במקור ל-GenerateDatabase תיאלץ לטפל בשגיאה, השאלה היא איך? לשיטה זו לא יהיה כל מושג איך לטפל בשגיאה שקרתה בעומק של מספק קריאות בנתיב הקוד. כפי שכבר ראינו, על השיטה הקוראת נאמר שאינה נמצאת בהקשר הנכון לטפל בשגיאה. במילים אחרות, השיטה היחידה שיכולה מבחינה לוגית ליצור מידע משמעותי כלשהו אודות השגיאה היא השיטה שנכשלה. עם זאת, אם משתמשים בקודים מוחזרים במחלקה AccessDatabase, כל שיטה בנתיב הקוד צריכה לבדוק כל קוד וקוד של שגיאה שכל שיטה אחרת עשויה להחזיר. בעיה אחת ברורה היא שהשיטה הקוראת צריכה לטפל פוטנציאלית במספר גדול ומגוחך של קודי שגיאה. בנוסף לכך, תחזוקת הקוד אינה קלה. בכל פעם שמוסיפים מצב של שגיאה לאחת מהשיטות בנתיב הקוד, יש לעדכן ביישום כל מופע אחר שבו שיטה קוראת לשיטה זו בכדי לטפל בקוד השגיאה החדש. מיותר לציין שהצעה זו אינה זולה במונחי עלות כוללת של בעלות על תוכנה (TCO).

טיפול בחריגים פותר את כל הסוגיות האלו בכך שהוא מאפשר לשיטה הקוראת ללכוד סוג נתון של חריג. בדוגמה שלנו, אם גוזרים מחלקה בשם AccessDatabaseException מ-Exception, ניתן להשתמש בה עבור כל סוגי השגיאות המתרחשות בכל אחת משיטות המחלקה AccessDatabase. (המחלקה Exception וגזירת מחלקות חריגים מותאמות יידונו מאוחר יותר בפרק בסעיף "השימוש במחלקה System.Exception").

אם השיטה CreateIndexes נכשלת, היא תבנה ותזרוק חריג מסוג AccessDatabaseException. השיטה הקוראת תתפוס חריג זה ותהיה מסוגלת לבחון את האובייקט Exception ולפענח מה בדיוק השתבש. לכן, במקום הצורך לטפל בכל סוג אפשרי של קוד שגיאה שהשיטה GenerateDatabase וכל אחת מהשיטות להן היא קוראת עשויות להחזיר, מובטח שאם כל אחת מהשיטות בנתיב קוד זה תיכשל, יוחזר לשיטה הקוראת המידע הנכון על השגיאה.

טיפול בחריגים מעניק בונוס נוסף: מאחר והמידע על השגיאה כלול בתוך מחלקה, מצבי שגיאה חדשים ניתנים להוספה ללא כל צורך לשנות את השיטה הקוראת. יכולת הרחבה זו - היכולת לבנות משהו ולאחר מכן להוסיף אליו בלי לשנות או לשבור את הקוד הקיים - היא אחת ההבטחות המקוריות של תכנות מוכוון-אובייקטים. מסיבות אלו, המושג של תפיסה וטיפול בשגיאות בהקשר הנכון הוא היתרון החשוב ביותר לשימוש בטיפול בחריגים.

## שיפור קריאות הקוד

כאשר משתמשים בקוד טיפול בחריגים, קריאות הקוד משתפרת לאין ערוך, תכונה זו נמצאת ביחס ישר להקטנת עלויות במונחים של תחזוקת הקוד. דרך הטיפול בקודים מוחזרים לעומת התחביר של טיפול בחריגים היא המקור לשיפור זה. אם היית משתמש בקודים מוחזרים עם השיטה AccessDatabase.GenerateDatabase שהוזכרה מקודם, היה נדרש קוד דומה לזה בכדי לטפל במצבי שגיאה:

```
public bool GenerateDatabase()
{
   if (CreatePhysicalDatabase())
   {
      if (CreateTables())
      {
         if (CreateIndexes())
         {
            return true;
         }
         else
         {
            // Handle error.
            return false;
         }
```

```
        }
        else
        {
            // Handle error.
            return false;
        }
    }
    else
    {
        // Handle error.
        return false;
    }
}
```

הוסף עוד מספר בדיקות אימות אחרות לקוד הקודם ואתה מסיים עם כמות עצומה של קוד לאימות שגיאות המעורב עם הלוגיקה העסקית שלך. אם בכל בלוק תשתמש בהזחת שורות קוד בגודל של ארבעה רווחים, התו הראשון של שורת קוד עשוי שלא להופיע החל מעמודה 20 ומעלה. אמנם אין בכך כדי להזיק לקוד עצמו, אך זה מקשה יותר על קריאתו ותחזוקתו. השורה התחתונה היא שקוד שהוא קשה לתחזוקה מהווה קרקע פורייה לשגיאות תוכנה. כך תיראה אותה הדוגמה אם היינו משתמשים בטיפול בחריגים:

```
// Calling code.
try
{
    AccessDatabase accessDb = new AccessDatabase();
    accessDb.GenerateDatabase();
}
catch(Exception e)
{
    // Inspect caught exception.
}

// Definition of AccessDatabase.GenerateDatabase method.
public void GenerateDatabase()
{
    CreatePhysicalDatabase();
    CreateTables();
    CreateIndexes();
}
```

שים לב עד כמה הפתרון האחרון נראה נקי ואלגנטי יותר מהקודם. זאת מהסיבה שהקוד לזיהוי השגיאה וטיפול בה אינו מעורב יותר עם הלוגיקה של הקוד הקורא לעצמו. היות וטיפול בחריגים הופך את הקוד לברור יותר, גם תחזוקתו נעשית קלה יותר.

## Throwing Exceptions From Constructors

יתרון גדול נוסף שיש לחריגים על פני טכניקות אחרות לטיפול בשגיאות היא בתחום של בניית האובייקט. היות ובנאי אינו יכול להחזיר ערכים, אין אמצעי קל ואינטואיטיבי לאותת לשיטה הקוראת לבנאי שהתרחשה שגיאה במהלך בניית האובייקט. עם זאת, ניתן להשתמש בחריגים מאחר והשיטה הקוראת רק צריכה לאחד (wrap) את בנאי האובייקט עם בלוק try, כפי שמדגים זאת הקוד הבא:

```
try
{
   // If the AccessDatabase object fails to construct
   // properly and throws an exception, it will now be caught.
   AccessDatabase accessDb = new AccessDatabase();
}
catch(Exception e)
{
   // Inspect the caught exception.
}
```

# השימוש במחלקה System.Exception

כפי שהזכרתי מקודם, כל חריג חייב להיות מהסוג (או נגזר מ-) System.Exception. המחלקה System.Exception היא למעשה מחלקת הבסיס של מספר מחלקות חריגים שניתן להשתמש בהם בקוד #C. רוב המחלקות שיורשות מ-System.Exception אינן מוסיפות פונקציונליות כלשהי למחלקת הבסיס. אם כך, מדוע צריך מחלקות נגזרות אם הן לא יהיו שונות באופן משמעותי ממחלקת הבסיס שלהן? הסיבה היא שלבלוק try יחיד יכולים להיות בלוקי catch מרובים כאשר כל בלוק catch מציין סוג חריג מסוים. כך הקוד מסוגל לטפל בחריגים שונים באופן המתאים לסוג מסוים של חריג.

## בניית אובייקט Exception

נכון לזמן כתיבת הספר, קיימים ארבעה בנאים (constructors) שונים עבור המחלקה System.Exception:

```
public Exception ();
public Exception(String);
protected Exception(SerializationInfo, StreamingContext);
public Exception(String, Exception);
```

הבנאי הראשון ברשימה זו הוא בנאי ברירת המחדל. הוא אינו מקבל ארגומנטים ופשוט קובע ערכי ברירת מחדל עבור כל המשתנים החברים. חריג זה נזרק בדרך כלל כך:

```
// Error condition reached.
throw new Exception();
```

הבנאי החריג השני מקבל ארגומנט יחיד של ערך מסוג String המזהה הודעת שגיאה, צורה שכבר ראית אותה ברוב הדוגמאות בפרק זה. הקוד שתופס את החריג מקבל את הודעת השגיאה באמצעות המאפיין System.Exception.Message. הנה דוגמה פשוטה לשני הצדדים של הפצת חריג זה:

```
using System;

class ThrowException3
{
   class FileOps
   {
      public void FileOpen(String fileName)
      {
         // ...
         throw new Exception("Oh bother");
      }

      public void FileRead()
      {
      }
   }

   public static void Main()
   {
      // Code catching exception.
      try
      {
         FileOps fileOps = new FileOps();

         fileOps.FileOpen("c:\\test.txt");
         fileOps.FileRead();
      }
      catch(System.Exception e)
      {
         Console.WriteLine(e.Message);
      }
   }
}
```

הבנאי השלישי יוצר מופע של המחלקה Exception עם נתונים רשומים.

הבנאי האחרון מאפשר להגדיר בנוסף להודעת שגיאה גם מה שידוע כ**חריג פנימי** (inner exception). הסיבה לכך היא שיש זמנים בהם רוצים לשנות מעט את החריג ברמה אחת של מחסנית הקריאות לפני שמעבירים אותו הלאה במעלה המחסנית. הדוגמה הבאה מראה מתי עשויים להשתמש בתכונה זו.

נניח שאתה רוצה להקל על לקוח של מחלקתך ואתה מחליט לזרוק רק סוג אחד של חריג. כך הלקוח צריך לתפוס חריג אחד בלבד ולפנות למאפיין InnerException של החריג. לזה יש גם תועלת נוספת הנובעת מכך שאם אתה מחליט לשנות שיטה נתונה באופן שתזרוק סוג חדש של חריג לאחר שקוד הלקוח כבר נכתב, אין צורך לעדכן את הלקוח אלא אם הוא רוצה לעשות משהו מסוים עם חריג זה.

בדוגמה הבאה, שים לב שקוד הלקוח תופס את החריג System.Exception של הרמה העליונה ומדפיס הודעה הכלולה באובייקט החריג הפנימי של חריג זה. אם בעתיד תזרוק השיטה DoWork סוגים אחרים של חריגים - כל זמן שהיא עושה זאת כחריגים פנימיים של אובייקט System.Exception - קוד הלקוח ימשיך לעבוד.

```
using System;
using System.Globalization;

class FooLib
{
   protected bool IsValidParam(string value)
   {
      bool success = false;

      if (value.Length == 3)
      {
         char c1 = value[0];
         if (Char.IsNumber(c1))
         {
            char c2 = value[2];
            if (Char.IsNumber(c2))
            {
               if (value[1] == '.')
                  success = true;
            }
         }
      }

      return success;
   }
```

```
    public void DoWork(string value)
    {
        if (!IsValidParam(value))throw new Exception
                ("", new FormatException("Invalid parameter specified"));
        Console.WriteLine("Work done with '{0}'", value);
    }
}

class FooLibClientApp
{
    public static void Main(string[] args)
    {
        FooLib lib = new FooLib();
        try
        {
            lib.DoWork(args[0]);
        }
        catch(Exception e)
        {
            Exception inner = e.InnerException;
            Console.WriteLine(inner.Message);
        }
    }
}
```

## השימוש במאפיין StackTrace

מאפיין שימושי אחר של המחלקה System.Exception הוא StackTrace. מאפיין זה מאפשר לך לקבוע - בכל נקודה נתונה שבה יש לך אובייקט System.Exception חוקי - איך נראית מחסנית הקריאות הנוכחית. התבונן בקוד הבא:

```
using System;

class StackTraceTestApp
{
    public void Open(String fileName)
    {
        Lock(fileName);
        // ...
    }
    public void Lock(String fileName)
```

```
    {
        // Error condition raised.
        throw new Exception("failed to lock file");
    }
    public static void Main()
    {
        StackTraceTestApp test = new StackTraceTestApp();

        try
        {
            test.Open("c:\\test.txt");
            // Use file.
        }
        catch(Exception e)
        {
            Console.WriteLine(e.StackTrace);
        }
    }
}
```

דוגמה זו מדפיסה את הפלט הבא:

```
at StackTraceTest.Main()
```

לפיכך, המאפיין StackTrace מחזיר את המחסנית בנקודה שבה נתפסה השגיאה. מאפיין זה יכול להיות שימושי בתסריטים של רישום יומן מעקב וניפוי שגיאות.

## איך לתפוס סוגי חריגים מרובים

יש מצבים שבהם רוצים שבלוק try יתפוס סוגי חריגים שונים. לדוגמה, שיטה נתונה יכולה להיות מתועדת כאחת הזורקת מספר סוגים שונים של חריגים, או שבלוק try אחד יכול להכיל מספר קריאות לשיטות שכל אחת מהן מתועדת כאחת שיכולה לזרוק סוג שונה של חריג. מתמודדים עם מצבים כאלה באמצעות הוספה של בלוק catch עבור כל סוג של חריג שהקוד צריך לטפל בו:

```
try
{
    Foo(); // Can throw FooException.
    Bar(); // Can throw BarException.
}
catch(FooException e)
{
    // Handle the error.
}
```

```
catch(BarException e)
{
   // Handle the error.
}

catch(Exception e)
{
}
```

כעת ניתן לטפל בכל סוג של חריג במסגרת בלוק catch נפרד (ולכן גם קוד לטיפול בשגיאות נפרד). עם זאת, פרט חשוב מאוד כאן הוא העובדה שמחלקת הבסיס מטופלת אחרונה. אם היינו מציבים בלוק catch זה בהתחלה, לא היה ניתן להגיע לבלוקים האחרים מאחר וכל החריגים נגזרים מ-System.Exception. באותה מידה, הקוד הבא יידחה על ידי המהדר:

```
try
{
   Foo(); // Can throw FooException.
   Bar(); // Can throw BarException.
}
catch(Exception e)
{
   // ***ERROR - THIS WON'T COMPILE
}
catch(FooException e)
{
   // Handle the error.
}
catch(BarException e)
{
   // Handle the error.
}
```

## גזירת מחלקות Exception מותאמות

כפי שאמרתי, לעיתים רוצים לספק מידע או עיצוב נוסף לחריג לפני שזורקים אותו לקוד הלקוח. אולי גם רוצים לספק מחלקת בסיס למצב זה כך שהמחלקה תפרסם את העובדה שהיא זורקת רק סוג אחד של חריג. בדרך זו, דאגתו היחידה של הלקוח היא לתפוס מחלקת בסיס זו.

דוגמה אחרת לצורך אפשרי בגזירת מחלקת Exception מותאמת היא מקרה שבו נזרק חריג בכל פעם שאתה רוצה לבצע פעולה, כגון רישום האירוע ביומן מעקב או שליחת דואר אלקטרוני למישהו במרכז תמיכה. במקרה זה אתה גוזר מחלקת Exception משלך ומכניס את הקוד הנדרש בבנאי המחלקה, הנה כך:

```
using System;

public class TestException : Exception

{
   // You would probably have extra methods and properties
   // here that augment the .NET Exception that you derive from.

   // Base Exception class constructors.
   public TestException()
      :base() {}
   public TestException(String message)
      :base(message) {}
   public TestException(String message, Exception innerException)
      :base(message, innerException) {}
}
public class DerivedExceptionTestApp
{
   public static void ThrowException()
   {
      throw new TestException("error condition");
   }
   public static void Main()
   {
      try
      {
         ThrowException();
      }
      catch(Exception e)
      {
         Console.WriteLine(e.ToString());
      }
   }
}
```

## גזירת מחלקות Exception

למרות שאין זה חוק ברזל, נוהג טוב בתכנות - ועקבי ברוב תוכניות #C שתראה - לכנות את מחלקות החריגים שלך בשמות המסתיימים במילה "Exception".

לדוגמה, אם תרצה לגזור מחלקת Exception עבור מחלקה הנקראת MyFancyGraphics, תוכל לקרוא למחלקה MyFancyGraphicsException.

כלל אצבע אחר, כאשר גוזרים מחלקות חריגים מותאמות אישית, הוא ליישם את כל בנאי המחלקה System.Exception (נכון לרגע זה יש 4). שוב, אין הדבר הכרחי, אך זה משפר את העקביות עם קוד #C אחר שהלקוח ישתמש בו.

הקוד הקודם ידפיס את הפלט הבא. שים לב שהקריאה לשיטה ToString מביאה להצגה של צירוף מאפיינים: הייצוג הטקסטואלי של שם מחלקת החריג, טקסט ההודעה המועבר לבנאי החריג והמאפיין StackTrace.

```
TestException: error condition
   at DerivedExceptionTestApp.Main()
```

# תכנון קוד עם טיפול בחריגים

עד כה סקרנו את המושגים הבסיסיים של טיפול בחריגים ואת הסמנטיקה המתייחסת לזריקה ותפיסה של חריגים. כעת נבחן היבט חשוב באותה מידה של טכניקה זו: להבין איך מתכננים מערכת שתשתמש בטיפול בחריגים. נניח שיש לך את השיטות Foo, Bar ו-Baz. השיטה Foo קוראת לשיטה Bar, אשר קוראת לשיטה Baz. אם Baz מתועדת כאחת הזורקת חריג, האם Bar צריכה לתפוס את החריג גם אם היא אינה מסוגלת או שאינה רוצה לעשות דבר עם חריג זה? איך צריך לפצל את הקוד ביחס לבלוקים try ו-catch?

## סוגיות תכנון של בלוק try

ידוע לך איך תופסים חריג ששיטה נקראת עשויה לזרוק. אתה גם יודע שהבקרה מועברת במעלה מחסנית הקריאות עד שנמצא בלוק catch מתאים. נשאלת השאלה, האם בלוק try צריך לתפוס כל חריג אפשרי ששיטה הכלולה בו עשויה לזרוק? התשובה היא לא, וזאת בכדי לשמור על היתרון הגדול ביותר של שימוש בטיפול בחריגים: צמצום קוד והוצאות תחזוקה קטנות יותר. הדוגמה הבאה ממחישה תוכנית שבה חריג שנתפס נזרק שוב כך שיטופל על ידי בלוק catch אחר:

```
using System;

class WhenNotToCatchApp
{
   public void Foo()
   {
      try
      {
         Bar();
      }
      catch(Exception e)
      {
         Console.WriteLine(e.Message);
      }
   }
```

```
    public void Bar()
    {
        try
        {

            Baz();
        }
        catch(Exception e)
        {
// Bar should not catch this because it
// doesn't do anything but rethrow it.

throw;
        }
    }

    public void Baz()
    {
        throw new Exception("Exception originally thrown by Baz");
    }

    public static void Main()
    {
        WhenNotToCatchApp test = new WhenNotToCatchApp();
        test.Foo(); // This method will ultimately
                    // print the error message.
    }
}
```

בדוגמה זו Bar תופסת את החריג ש-Baz זורקת למרות שאינה עושה דבר מלבד לזרוק אותו שוב אחורה ל-Foo. לאחר מכן Foo תופסת את החריג ועושה משהו עם המידע - במקרה זה, מציגה את המאפיין הודעת שגיאה של האובייקט Exception. הנה מספר סיבות מדוע שיטות שסתם כך זורקות הלאה חריג אינן צריכות לתפוס חריג זה מלכתחילה:

- מאחר והשיטה אינה עושה דבר עם החריג, אתה מגדיר בלוק catch מיותר במקרה הטוב ביותר. ברור ללא ספק שאין טעם בקיומו של קוד שאין לו שום תפקיד לחלוטין.
- אם סוג החריג ש-Baz זורקת היה משתנה בעתיד, היית צריך לחזור אחורה ולשנות את הבלוקים catch של השיטות Bar ו-Foo גם יחד. אין לך כל סיבה להכניס את עצמך למצב שבו יהיה עליך לשנות קוד שממילא אינו עושה דבר.

מאחר ו-CLR ממשיכה אוטומטית במעלה מחסנית הקריאות עד ששיטה תופסת את החריג, שיטות ביניים יכולות להתעלם מהחריגים שהן אינן מסוגלות לעבד. עליך רק להבטיח ששיטה כלשהי תתפוס את החריג, אחרת, כפי שהזכרתי מקודם, אם נזרק חריג ולא נמצא בלוק catch במחסנית הקריאות הנוכחית, ביצוע היישום יופסק.

## סוגיות תכנון של בלוק catch

הקוד היחיד שצריך להופיע בבלוק catch הוא כזה שיעבד לפחות חלקית את החריג הנתפס. לדוגמה, קורה ששיטה תופסת חריג, עושה את אשר ביכולה בכדי לעבד את החריג ואז זורקת אותו שוב כדי לאפשר טיפול נוסף בשגיאה. הדוגמה הבאה ממחישה זאת. Foo קוראת ל-Bar, אשר מבצעת עבודת מסד נתונים כלשהי לטובת Foo. היות ומשתמשים בבקרת מחויבות בעדכון מסד הנתונים, או בתנועות (transactions), Bar צריכה לתפוס כל שגיאה שמתרחשת ולבצע גלילה לאחור, כלומר ביטול התנועות שלא הושלמו, לפני שהיא זורקת שוב את החריג בחזרה ל-Foo.

```
using System;

class WhenToCatchApp
{
   public void Foo()
   {
      try
      {
         Bar();
      }
      catch(Exception e)
      {
         Console.WriteLine(e.Message);
      }
   }

   public void Bar()
   {
      try
      {
         // Call method to set a "commitment boundary."
         Console.WriteLine("setting commitment boundary");

         // Call Baz to save data.
         Console.WriteLine("calling Baz to save data");
         Baz();

         Console.WriteLine("commiting saved data");
      }
```

```
        catch(Exception)
        {
            // In this case, Bar should catch the exception
            // because it's doing something significant
            //(rolling back uncommitted database changes).

            Console.WriteLine("rolling back uncommited changes " +

                "and then rethrowing exception");

          throw;
        }
    }

    public void Baz()
    {
        throw new Exception("db failure in Baz");
    }

    public static void Main()
    {
        WhenToCatchApp test = new WhenToCatchApp();
        test.Foo(); // This method will ultimately print the error msg.
    }
}
```

bar היתה צריכה לתפוס את החריג בכדי לבצע עיבוד שגיאה משלה. כאשר היא סיימה, היא זרקה שוב את החריג בחזרה למחסנית הקריאות, שם הוא נתפס על ידי Foo אשר היתה מסוגלת לבצע עיבוד שגיאה משלה. היכולת של כל רמה במחסנית הקריאות לבצע רק את מידת עיבוד השגיאה שאותו היא יכולה לבצע בהתאם להקשר שלה, היא דוגמה אחת הממחישה מדוע הטיפול בחריגים הוא כה חשוב בתכנון המערכת שלך.

# סיכום

תחביר #C של טיפול בחריגים הוא פשוט ומובן מאליו. מימוש טיפול בחריגים ביישומים שלך הוא קל כמו לתכנן את שיטותיך מראש. היעד העיקרי של NET CLR הוא לעזור במניעת שגיאות זמן-ריצה באמצעות CTS ולטפל במסגרת בכל שגיאת זמן-ריצה שמתרחשת. השימוש בקוד טיפול בחריגים ביישומים שלך הופך אותם לחסונים ומהימנים יותר, ובסופו של דבר לקלים יותר לתחזוקה.

פרק 13

# העמסת אופרטורים והמרות מותאמות - Operator Overloading and User-Defined Conversions

בפרק 7 למדת איך להשתמש באופרטור [] בהגדרת מחלקה בכדי לגשת לאובייקט באמצעות אינדקסים כאילו היה מערך. בפרק זה נבחן שתי תכונות קרובות של #C המאפשרות ליצור ממשקים של מחלקות ומבנים שהם קלים ואינטואיטיביים יותר לשימוש: העמסת אופרטורים (operator overloading) והמרות מותאמות אישית (user-defined conversions). נתחיל בסקירה כללית על העמסת אופרטורים במונחי היתרונות שהיא מעניקה. לאחר מכן נלמד איך מגדירים מחדש את התנהגויות ברירת המחדל של אופרטורים ונסתכל על דוגמה מציאותית שבה מעמיסים את האופרטור + בכדי לקבץ אובייקטים מרובים מסוג Invoice. בהמשך נלמד אילו אופרטורים בינאריים ואונאריים ניתנים להעמסה וגם על חלק מההגבלות הכרוכות בכך. לסיום אציג מספר קווים מנחים לתכנון שיש להתחשב בהם כאשר אתה שוקל העמסת אופרטורים עבור המחלקות שלך. בתום הדיון בהעמסת אופרטורים נעסוק במושג חדש הנקרא המרות מותאמות אישית. תחילה אסביר את יסודות התכונה ולאחר מכן אעבור לדוגמה של מחלקה הממחישה איך להשתמש בהמרות בכדי לאפשר את ההמרה של struct או class ל-struct או class אחר, או לסוג בסיסי של #C.

## Operator Overloading

העמסת אופרטורים מאפשרת להגדיר מחדש אופרטורים קיימים ב-#C לשימוש עם סוגים מותאמים. העמסת אופרטורים כונתה "סוכר סינתטי" בזכות העובדה שהעמסת אופרטור היא פשוט אמצעי אחר לקרוא לשיטה. על תכונה זאת גם נאמר שבאופן בסיסי אינה מוסיפה דבר לשפה. למרות שזה נכון מבחינה טכנית, העמסת אופרטורים אכן עוזרת באחד מההיבטים החשובים של תכנות מוכוון-אובייקטים: הפשטה.

נניח שברצונך לקבץ אוסף של חשבוניות עבור לקוח מסוים. באמצעות העמסת אופרטורים תוכל לכתוב קוד שבו האופרטור =+ מועמס, בדומה לזה:

```
Invoice summaryInvoice = new Invoice();
foreach (Invoice invoice in customer.GetInvoices())
{
   summaryInvoice += invoice;
}
```

היתרונות בקוד כמו זה הם תחביר אינטואיטיבי וכן הפשטה, מעצם העובדה שהלקוח אינו נדרש להבין את פרטי המימוש של איך לקבץ את החשבוניות. במילים פשוטות, העמסת אופרטורים תומכת ביצירת תוכנה שהיא פחות יקרה לכתיבה ולתחזוק.

## תחביר ודוגמה

כפי שאמרתי, העמסת אופרטורים היא אמצעי אחר לקרוא לשיטה. בכדי להגדיר מחדש אופרטור עבור מחלקה, עליך להשתמש בתבנית הבאה, שבה op הוא האופרטור שאותו מעמיסים:

**public static** retval **operator**op ( object1 [, object2 ])

עליך להיות מודע לעובדות הבאות כאשר אתה משתמש בהעמסת אופרטור:

- כל השיטות של אופרטור מועמס חייבות להיות מוגדרות public או static.
- מבחינה טכנית, הערך המוחזר (retval) יכול להיות מכל סוג. עם זאת, מקובל להחזיר את הסוג עבורו השיטה מוגדרת, חוץ מהאופרטורים true ו-false אשר תמיד מחזירים ערך בוליאני.
- מספר הארגומנטים המועברים (object1, object2) תלוי בסוג האופרטור המועמס. אם מועמס אופרטור אונארי ( אופרטור בעל אופרנד יחיד), מועבר ארגומנט אחד. אם מועמס אופרטור בינארי (אופרטור המקבל שני אופרנדים), מועברים שני ארגומנטים.
- במקרה של אופרטור אונארי, הארגומנט עבור השיטה חייב להיות מאותו סוג כמו זה של המבנה או המחלקה הסוגר(ת). במילים אחרות, אם מגדירים מחדש את האופרטור ! עבור מחלקה בשם Foo, שיטה זו חייבת לקבל ארגומנט יחיד של משתנה מסוג Foo.
- אם מועמס אופרטור בינארי, הארגומנט הראשון חייב להיות מאותו הסוג כמו זה של המחלקה הסוגרת והארגומנט השני יכול להיות מכל סוג.

בפסידו-קוד המוצג בסעיף קודם השתמשתי באופרטור =+ עבור המחלקה Invoice. מסיבות שאותם תבין בקרוב, לא ניתן למעשה להעמיס אופרטורים מורכבים אלה. ניתן להעמיס רק את אופרטור "הבסיס" - במקרה זה, האופרטור +. הנה התחביר המשמש להגדרת השיטה operator+ עבור המחלקה Invoice:

```
public static Invoice operator+ (Invoice invoice1, Invoice invoice2)
{
   // Create a new Invoice object.
   // Add the desired contents from
   // invoice1 to the new Invoice object.
   // Add the desired contents from
   // invoice2 to the new Invoice object.
   // Return the new Invoice object.
 }
```

כעת נתבונן בדוגמה מוצקה יותר, הכוללת שתי מחלקות: Invoice ו-InvoiceDetailLine. המחלקה Invoice מכילה משתנה חבר מסוג ArrayList המייצג אוסף של כל שורות הפירוט בחשבונית. בכדי לאפשר צבירה של כל שורות הפירוט עבור חשבוניות מרובות העמסתי את האופרטור +. השיטה Invoice.operator+ יוצרת אובייקט Invoice חדש וניגשת לכל אלמנט במערכי שורות פירוט חשבונית של שני האובייקטים Invoice המועברים כארגומנטים. אובייקט Invoice זה מוחזר לאחר מכן לשיטה הקוראת. ברור שמודול הפקת חשבוניות אמיתי יהיה יותר מורכב מזה. כאן הכוונה להראות באופן מציאותי במידת מה איך ניתן להשתמש בהעמסת אופרטורים.

```
using System;
using System.Collections;

class InvoiceDetailLine
{
   double lineTotal;
   public double LineTotal
   {
      get
      {
         return this.lineTotal;
      }
   }

   public InvoiceDetailLine(double LineTotal)
   {
      this.lineTotal = LineTotal;
   }
}
```

```
class Invoice
{
   public ArrayList DetailLines;

   public Invoice()
   {
      DetailLines = new ArrayList();
   }

   public void PrintInvoice()
   {
      Console.WriteLine("\nLine Nbr\tTotal");

      int i = 1;
      double total = 0;
      foreach(InvoiceDetailLine detailLine in DetailLines)
      {
         Console.WriteLine("{0}\t\t{1}", i++, detailLine.LineTotal);
         total += detailLine.LineTotal;
      }

      Console.WriteLine("=====\t\t===");
      Console.WriteLine("Total\t\t{1}", i++, total);
   }

   public static Invoice operator+ (Invoice invoice1, Invoice invoice2)
   {
      Invoice returnInvoice = new Invoice();

      foreach (InvoiceDetailLine detailLine in invoice1.DetailLines)
      {
         returnInvoice.DetailLines.Add(detailLine);
      }

      foreach (InvoiceDetailLine detailLine in invoice2.DetailLines)
      {
         returnInvoice.DetailLines.Add(detailLine);
      }
      return returnInvoice;
   }
}
```

```
class InvoiceAddApp
{
   public static void Main()
   {
      Invoice i1 = new Invoice();
      for (int i = 0; i < 2; i++)
      {
         i1.DetailLines.Add(new InvoiceDetailLine(i + 1));
      }

      Invoice i2 = new Invoice();
      for (int i = 0; i < 2; i++)
      {
         i2.DetailLines.Add(new InvoiceDetailLine(i + 1));
      }

      Invoice summaryInvoice = i1 + i2;

      summaryInvoice.PrintInvoice();
   }
}
```

## האופרטורים שניתן להעמיס

ניתן להעמיס רק את האופרטורים האונאריים והבינאריים הבאים.

אופרטורים אונאריים: +, -, !, ~, ++, --, true, false

אופרטורים בינאריים: +, -, *, /, %, &, |, ^, <<, >>, ==, !=, >, <, >=, <=

כאן משמש הפסיק כתו מפריד בין האופרטורים השונים הניתנים להעמסה. האופרטור פסיק, שבו משתמשים במשפט for ובקריאות לשיטות, אינו ניתן להעמסה.

## הגבלות על העמסת אופרטורים

אין אפשרות להעמיס את אופרטור ההצבה =. יחד עם זאת, כאשר מעמיסים אופרטור בינארי, ההצבה המורכבת השקולה שלו מועמסת באופן משתמע. לדוגמה, אם מעמיסים את האופרטור +, האופרטור =+ מועמס באופן משתמע בכך שהשיטה המותאמת operator+ נקראת.

לא ניתן להעמיס את האופרטורים []. עם זאת, כפי שראית בפרק 7, הגדרת אובייקט מותאם עם גישה דמויית מערך מתאפשרת באמצעות סדרנים.

גם הסוגריים המשמשים לביצוע המרה אינם ניתנים להעמסה. במקום זה יש להשתמש באופרטורים של המרה, אשר מתייחסים אליהם גם כאל המרות מותאמות. נושא זה יידון בחלק השני של הפרק.

לא ניתן להעמיס אופרטורים שכרגע אינם מוגדרים בשפת #C. לדוגמה, לא ניתן להגדיר את ** כאמצעי להגדרת פעולות של העלאה בחזקה, מאחר ו-#C אינה מגדירה אופרטור **. בנוסף לכך, לא ניתן לשנות תחביר של אופרטור. לא ניתן לשנות את האופרטור הבינארי * כך שיקבל שלושה ארגומנטים בזמן שהתחביר שלו (על פי ההגדרה) קורא לשני אופרנדים. לבסוף, לא ניתן לשנות את הקדימות של אופרטור. חוקי הקדימות קבועים - עיין בפרק 10 למידע נוסף על חוקים אלה.

## קווים מנחים לתכנון

לאחר שראית מהי העמסת אופרטורים ואיך להשתמש בה ב-#C, נעסוק כעת באחד ההיבטים של תכונה שימושית זו, אשר לעיתים קרובות מתעלמים ממנו: קווים מנחים לתכנון. יש להתרחק מן הנטייה הטבעית לרצות להשתמש בתכונה חדשה לשם השימוש בה. לתופעה זו מתייחסים לעיתים כאל "פתרון המחפש בעיה". אולם גישת תכנון טובה היא תמיד לזכור את המימרה "קוד נקרא יותר מאשר הוא נכתב". חשוב על מחלקת הלקוח בבואך לקבוע אם ומתי להעמיס אופרטור או קבוצה של אופרטורים. הנה כלל אצבע: אתה צריך להעמיס אופרטור רק אם זה הופך את ממשק המחלקה לאינטואיטיבי יותר לשימוש. לדוגמה, יש היגיון מושלם בלהיות מסוגל לחבר יחד חשבוניות.

בנוסף לאמור לעיל, אל תשכח שעליך לחשוב כמו לקוח של מחלקתך. נניח שאתה כותב מחלקת Invoice ואתה רוצה שהלקוח יוכל לתת הנחה על סכום החשבונית. אני ואתה עשויים לדעת שפריט שורת הנחה יתווסף לחשבונית, אך הנקודה בכימוס היא שלקוחותיך אינם צריכים לדעת את פרטי היישום המדויקים של מחלקתך. לפיכך, העמסת האופרטור * - כמוצג כאן - עשויה להיות רעיון טוב מאחר והיא עוזרת להפוך את ממשק המחלקה Invoice לטבעי ואינטואיטיבי:

```
invoice *= .95; // 5% discount.
```

# User-Defined Conversions

ציינתי כבר שלא ניתן להעמיס את הסוגריים המשמשים להמרה ובמקום זה יש להשתמש בהמרות מותאמות. המרות מותאמות מאפשרות להגדיר המרות עבור מבנים או מחלקות כך שאותו struct או class ניתנים להמרה למבנים, מחלקות או סוגי #C בסיסיים אחרים.

מדוע ומתי תרצה לעשות כך? נניח שביישום שלך אתה צריך להשתמש בסקאלות הטמפרטורה הסטנדרטיות של צלסיוס ופרנהייט כך שניתן להמיר בקלות בין השתיים. תוכל להשתמש בתחביר הבא ליצירת המרות מותאמות:

```
Fahrenheit f = 98.6F;
Celsius c = (Celsius)f; // User-defined conversion.
```

תחביר זה אינו מספק יתרון פונקציונלי כלשהו על פני התחביר הבא, אך הוא אינטואיטיבי יותר לכתיבה וקל יותר לקריאה.

```
Fahrenheit f = 98.6F;
Celsius c = f.ConvertToCelsius();
```

## תחביר ודוגמה

התחביר של הצהרה על המרות מותאמות משתמש במילת המפתח operator:

**public static implicit operator** conv-type-out (conv-type-in operand)

**public static explicit operator** conv-type-out (conv-type-in operand)

יש מספר חוקים בנוגע לתחביר של הגדרת המרות:

- כל שיטת המרה עבור struct או class - ניתן להגדיר שיטות כאלו ככל שנדרש - חייבת להיות static.
- המרות חייבות להיות מוגדרות implicit או explicit. מילת המפתח implicit אומרת שהלקוח אינו נדרש לבצע את ההמרה והיא תתרחש באופן אוטומטי. לעומת זאת, השימוש במילת המפתח explicit מציין שהלקוח חייב להמיר את הערך לסוג הרצוי בצורה מפורשת.
- כל ההמרות חייבות לקבל (כארגומנט) את הסוג שההמרה מוגדרת עליו או שהן חייבות להחזיר סוג זה.
- כמו בהעמסת אופרטורים, החתימה של שיטת המרה עושה שימוש במילת המפתח operator אך ללא תוספת סימן אופרטור כלשהו.

ניעזר בדוגמה להבהרת חוקים אלה. דוגמה זו כוללת שני מבנים (Celsius ו-Farenheit) המאפשרים ללקוח להמיר ערך מסוג float לאחת מסקאלות הטמפרטורה. תחילה אציג את המבנה Celsius ואסביר מספר נקודות בנוגע אליו. לאחר מכן אציג את היישום בשלמותו.

```
struct Celsius
{
   public Celsius(float temp)
   {
      this.temp = temp;
   }

   public static implicit operator Celsius(float temp)
   {
      Celsius c;
      c = new Celsius(temp);
      return(c);
   }

   public static implicit operator float(Celsius c)
   {
      return((((c.temp - 32) / 9) * 5));
   }

   public float temp;
}
```

ההחלטה הראשונה שאתה רואה היא להשתמש במבנה במקום במחלקה. לא היתה לי כל סיבה ממשית לעשות כן פרט לעובדה שהשימוש במחלקות הוא יקר יותר מהשימוש במבנים - במונחים של אופן הקצאת מחלקות - ומחלקה באמת אינה נחוצה כאן מאחר והמבנה Celsius אינו זקוק לתכונות #C ייחודיות של מחלקה, כגון הורשה.

לאחר מכן, שים לב שהצהרתי על בנאי המקבל ארגומנט בודד של ערך מסוג float. ערך זה מאוחסן במשתנה חבר בשם temp. כעת הסתכל על אופרטור ההמרה המוגדר מייד לאחר הבנאי של המבנה. שיטה זו היא אשר נקראת כאשר הלקוח מנסה להמיר Float ל-Celsius או כאשר, כמו עם שיטה, הלקוח משתמש בערך float היכן שהתוכנית מצפה לקבל מבנה Celsius. שיטה זו אינה עושה הרבה, ולמעשה היא קוד נוסחתי שניתן להשתמש בו ברוב ההמרות הבסיסיות. כאן אני פשוט יוצר מופע של מבנה Celsius ואז מחזיר מבנה זה על ידי הקריאה return. קריאה זו היא אשר גורמת לקריאה לשיטה האחרונה המוגדרת במבנה. כפי שרואים, שיטה זו פשוט מספקת נוסחה מתמטית עבור המרה מערך פרנהייט לערך צלסיוס.

הנה קוד היישום המלא, כולל המבנה Farenheit:

```
using System;

struct Celsius
{
   public Celsius(float temp)
```

```
    {
        this.temp = temp;
    }

    public static implicit operator Celsius(float temp)
    {
        Celsius c;
        c = new Celsius(temp);
        return(c);
    }

    public static implicit operator float(Celsius c)
    {
        return((((c.temp - 32) / 9) * 5));
    }

    public float temp;
}

struct Fahrenheit
{
    public Fahrenheit(float temp)
    {
        this.temp = temp;
    }

    public static implicit operator Fahrenheit(float temp)
    {
        Fahrenheit f;
        f = new Fahrenheit(temp);
        return(f);
    }

    public static implicit operator float(Fahrenheit f)
    {
        return((((f.temp * 9) / 5) + 32));
    }

    public float temp;
}

class Temp1App
{
    public static void Main()
```

```
    {
        float t;

        t=98.6F;
        Console.Write("Conversion of {0} to Celsius = ", t);
        Console.WriteLine((Celsius)t);

        t=0F;
        Console.Write("Conversion of {0} to Fahrenheit = ", t);
        Console.WriteLine((Fahrenheit)t);
    }
}
```

הנה הפלט המתקבל לאחר הידור וביצוע היישום:

```
Conversion of 98.6 to Celsius = 37
Conversion of 0 to Fahrenheit = 32
```

יישום זה פועל די טוב, והיכולת לכתוב (Celsius)98.6F היא בוודאות אינטואיטיבית יותר מלקרוא לשיטה כלשהי של מחלקה סטטית. עם זאת, שים לב שניתן להעביר רק ערכים מסוג float לשיטות המרה אלו. לכן הקוד הבא לא יעבוד עבור היישום הקודם:

```
Celsius c = new Celsius(55);
Console.WriteLine((Fahrenheit)c);
```

בנוסף לכך, היות ואין שיטת המרה Celsius הלוקחת מבנה מסוג Fahrenheit או ההיפך, הקוד צריך להניח שהערך המועבר הוא הערך שיש להמירו. במילים אחרות, אם קוראים ל- (Celsius)98.6F מקבלים את הערך 37. אולם, אם לאחר מכן ערך זה מועבר שוב לשיטת ההמרה, אין לשיטה זו כל דרך לדעת שהערך כבר עבר המרה ומבחינה לוגית ערך זה כבר מייצג טמפרטורת צלסיוס חוקית - עבור שיטת ההמרה זהו רק ערך float. כתוצאה מכך, הערך שוב עובר המרה. לפיכך, עלינו לשנות את היישום כך שכל מבנה יוכל לקבל את המבנה האחר כארגומנט חוקי.

למרות שבהתחלה חששתי מהקושי לבצעה, התברר בדיעבד שמשימה זו היתה קלה ביותר. הנה הקוד המתוקן:

```
using System;

class Temperature
{
    public Temperature(float Temp)
    {
        this.temp = Temp;
    }
```

```
    protected float temp;
    public float Temp
    {
        get
        {
            return this.temp;
        }
    }
}

class Celsius : Temperature
{
    public Celsius(float Temp)
        : base(Temp) {}

    public static implicit operator Celsius(float Temp)
    {
        return new Celsius(Temp);
    }

    public static implicit operator Celsius(Fahrenheit F)
    {
        return new Celsius(F.Temp);
    }

    public static implicit operator float(Celsius C)
    {
        return((((C.temp - 32) / 9) * 5));
    }
}

class Fahrenheit : Temperature
{
    public Fahrenheit(float Temp)
        : base(Temp) {}

    public static implicit operator Fahrenheit(float Temp)
    {
        return new Fahrenheit(Temp);
    }

    public static implicit operator Fahrenheit(Celsius C)
    {
        return new Fahrenheit(C.Temp);
    }
```

```
    public static implicit operator float(Fahrenheit F)

    {
        return((((F.temp * 9) / 5) + 32));
    }
}

class Temp2App
{
    public static void DisplayTemp(Celsius Temp)
    {
        Console.Write("Conversion of {0} {1} to Fahrenheit = ",
            Temp.ToString(), Temp.Temp);
        Console.WriteLine((Fahrenheit)Temp);
    }

    public static void DisplayTemp(Fahrenheit Temp)
    {
        Console.Write("Conversion of {0} {1} to Celsius = ",
            Temp.ToString(), Temp.Temp);
        Console.WriteLine((Celsius)Temp);
    }

    public static void Main()
    {
        Fahrenheit f = new Fahrenheit(98.6F);
        DisplayTemp(f);

        Celsius c = new Celsius(0F);
        DisplayTemp(c);
    }
}
```

הדבר הראשון שיש לציין הוא ששיניתי את הסוגים Celsius ו-Fahrenheit ממבנים למחלקות. עשיתי זאת כדי שיהיו לנו שתי דוגמאות - אחת המשתמשת ב-struct ואחת המשתמשת ב-class. סיבה מעשית יותר לשינוי היא לשתף את המשתנה החבר temp על ידי גזירת המחלקות Celsius ו-Fahrenheit מאותה מחלקת בסיס Temperature. כעת ניתן גם להשתמש בשיטה ToString שעברה בהורשה (מ-System.Object) עבור הפלט של היישום.

ההבדל הנוסף היחיד שיש לציין הוא הוספת המרה עבור כל אחת מסקאלות הטמפרטורה, כאשר כל המרה מקבלת כארגומנט ערך של סקאלת הטמפרטורה האחרת.

שים לב לדמיון בקוד של שתי שיטות ההמרה:

```
public static implicit operator Celsius(float temp)
{
   Celsius c;
   c = new Celsius(temp);
   return(c);
}

public static implicit operator Celsius(Fahrenheit f)
{
   Celsius c;
   c = new Celsius(f.temp);
   return(c);
}
```

המשימה היחידה שנדרשתי לבצע היתה לשנות את הארגומנט המועבר, ולקרוא את הטמפרטורה מהאובייקט המועבר במקום ערך מסוג float בקידוד קשיח. מסיבה זו ציינתי מקודם עד כמה קלות ונוסחתיות הן שיטות המרה מרגע שאתה מבין את היסודות.

# סיכום

העמסת אופרטורים והמרות מותאמות שימושיים ביצירת ממשקים אינטואיטיביים עבור מחלקות. כאשר משתמשים בהעמסת אופרטורים יש להיות מודע להגבלות הנוגעות לכך בתכנון מחלקות. לדוגמה, בעוד שלא ניתן להעמיס את אופרטור ההצבה =, כאשר מעמיסים אופרטור בינארי אזי ההצבה המורכבת השקולה שלו מועמסת באופן משתמע. נהג על פי הקווים המנחים לתכנון בבואך להחליט מתי להשתמש בכל אחת מהתכונות. חשוב כמו לקוח של מחלקתך כאשר אתה מחליט באם להעמיס אופרטור או קבוצת אופרטורים. עם מעט ראייה חודרת לאופן השימוש של לקוחות במחלקותיך, תוכל להשתמש בתכונות רבות עוצמה אלו בכדי להגדיר את המחלקות כך שניתן יהיה לבצע פעולות מסוימות עם תחביר טבעי יותר.

פרק 14

# נציגים ומטפלים באירועים - Delegates and Event Handlers

חידוש שימושי אחר של שפת #C הם **נציגים** (delegates), המשרתים ביסודו של דבר אותה מטרה כמו של מצביעי פונקציות ב- ++C. עם זאת, נציגים הם אובייקטים מבטיחי-סוג (type-safe), מנוהלים ומאובטחים. הכוונה היא שסביבת זמן הריצה מבטיחה שנציג מצביע תמיד לשיטה תקפה, ולכן השימוש בנציגים מאפשר ליהנות מכל היתרונות של מצביעי פונקציות ללא הסכנות הכרוכות בכך, כמו כתובות שאינן תקפות או נציג הדורס את הזיכרון המוקצה לאובייקטים אחרים. בפרק זה נלמד על נציגים, נשווה אותם לממשקים, נלמד את התחביר המשמש להגדרתם ומהם הבעיות השונות שנציגים תוכננו בכדי לטפל בהם. כמו כן נראה מספר דוגמאות של שימוש בנציגים עם שיטות משוב וטיפול אסינכרוני באירועים גם יחד.

בפרק 9 ראינו איך ממשקים מוגדרים ומיושמים ב-#C. כזכור לך, מבחינה מושגית ממשקים הם פשוט חוזה בין שני קטעי קוד נפרדים. עם זאת, ממשקים די דומים למחלקות בכך שהגדרתם נעשית בזמן ההידור והם יכולים להכיל שיטות, מאפיינים, סדרנים ואירועים. לעומת זאת, נציגים מתייחסים רק לשיטות יחידות ומוגדרים בזמן ריצה. לנציגים יש שני שימושים עיקריים בתכנות בשפת #C: שיטות משוב וטיפול באירועים. אנו נעסוק תחילה בשיטות משוב.

## שימוש בנציגים כשיטות משוב

**שיטות משוב** (callback methods) מנוצלות במידה ניכרת בתכנות עבור Microsoft Windows. משתמשים בשיטות משוב כאשר צריך להעביר מצביע פונקציה לפונקציה אחרת אשר לאחר מכן **קוראת חזרה** (באמצעות המצביע המועבר) ומספקת מידע. דוגמה לכך היא הפונקציה EnumWindows של Win32 API.

פונקציה זו מונה את כל חלונות הרמה העליונה המוצגים על המסך והיא קוראת לפונקציית משוב עבור כל חלון ברשימת החלונות שלה. קריאות חוזרות משרתות מטרות רבות. להלן המטרות השכיחות ביותר:

- **עיבוד אסינכרוני** - שיטות משוב משמשות בעיבוד אסינכרוני כאשר לקוד הנקרא לוקח זמן רב לעבד את המבוקש. בדרך כלל התסריט פועל כך: קוד לקוח קורא לשיטה ומעביר לה את שיטת המשוב. השיטה הנקראת מפעילה מטלה (thread) ומיד חוזרת. לאחר מכן מבצעת המטלה את רוב העבודה תוך שהיא קוראת לשיטת המשוב במידת הצורך. היתרון הברור הוא בכך שהלקוח יכול להמשיך את העיבוד מבלי להתעכב בגין קריאה סינכרונית שעשויה לקחת זמן רב.

- **השתלת קוד מותאם לתוך נתיב קוד מחלקה** - שימוש נפוץ אחר לשיטות משוב הוא כאשר מחלקה מאפשרת ללקוח להגדיר שיטה הנקראת כדי לבצע עיבוד מותאם. נמחיש זאת בעזרת דוגמה מ-Windows. המחלקה ListBox ב-Windows מאפשרת לך להגדיר את מיון הפריטים בסדר עולה או יורד. מלבד מספר אפשרויות מיון בסיסיות אחרות, המחלקה ListBox אינה יכולה לספק לך מרחב פעולה של ממש והיא עדיין נשארת בגדר מחלקה כללית. לפיכך, המחלקה ListBox מאפשרת לך גם להגדיר פונקציית משוב עבור מיון. בדרך זו, כאשר ListBox ממיינת את הפריטים היא קוראת לפונקציית המשוב והקוד שלך יכול לבצע את המיון המותאם הדרוש לך.

כעת ניעזר בדוגמה בכדי ללמוד איך להגדיר ולהשתמש בנציג. בדוגמה זו יש מחלקת מנהל מסד נתונים העוקבת אחר כל החיבורים הפעילים למסד הנתונים ומספקת שיטה למניית חיבורים אלה. בהנחה שמנהל מסד הנתונים נמצא בשרת מרוחק, תהיה זו החלטת תכנון טובה להפוך את השיטה לאסינכרונית ולאפשר ללקוח לספק שיטת משוב. במציאות כותבים בדרך כלל יישום עם ריבוי מטלות בכדי שהפעולה תהיה באמת אסינכרונית. עם זאת, לשם פשטות הדוגמה - וגם מכיון שטרם עסקנו בנושא זה - לא נשתמש כאן בריבוי מטלות.

תחילה נגדיר שתי מחלקות ראשיות: DBManager ו-DBConnection:

```
class DBConnection
{
   .
   .
   .
}
```

```
class DBManager
{
  static DBConnection[] activeConnections;
        .
        .
        .

  public delegate void EnumConnectionsCallback(DBConnection connection);
    public static void EnumConnections(EnumConnectionsCallback callback)
    {
        foreach (DBConnection connection in activeConnections)
        {
            callback(connection);
        }
    }
}
```

השיטה EnumConnectionsCallback היא הנציג והיא מוגדרת על ידי הצבת מילת המפתח delegate בקדמת החתימה של השיטה. שיטה זו מוגדרת כמחזירה void ומקבלת ארגומנט יחיד: אובייקט DBConnection. לאחר מכן מוגדרת השיטה EnumConnections כמקבלת ארגומנט יחיד של השיטה EnumConnectionsCallback. בכדי לקרוא לשיטה DBManager.EnumConnections צריך רק להעביר לה מופע של הנציג DBManager.EnumConnectionCallback.

לשם כך, מחדשים (new) את הנציג עם שם השיטה בעלת אותה חתימה כמו של הנציג. הנה דוגמה:

```
DBManager.EnumConnectionsCallback myCallback =
    new DBManager.EnumConnectionsCallback(ActiveConnectionsCallback);

DBManager.EnumConnections(myCallback);
```

שים לב שניתן לאחד את שתי השורות לכדי קריאה אחת, הנה כך:

```
DBManager.EnumConnections(new
   DBManager.EnumConnectionsCallback(ActiveConnectionsCallback));
```

זה הכל באשר לתחביר הבסיסי של נציגים. כעת נסתכל על יישום הדוגמה המלא:

```
using System;

class DBConnection
{
   public DBConnection(string name)
   {
      this.name = name;
   }
```

```
    protected string Name;
    public string name
    {
        get
        {
            return this.Name;
        }
        set
        {
            this.Name = value;
        }
    }
}

class DBManager
{
    static DBConnection[] activeConnections;
    public void AddConnections()
    {
        activeConnections = new DBConnection[5];
        for (int i = 0; i < 5; i++)
        {
            activeConnections[i] =
new DBConnection("DBConnection " + (i + 1));
        }
    }

    public delegate void EnumConnectionsCallback(DBConnection connection);
    public static void EnumConnections(EnumConnectionsCallback callback)
    {
        foreach (DBConnection connection in activeConnections)
        {
            callback(connection);
        }
    }
}

class Delegate1App
{
    public static void ActiveConnectionsCallback(DBConnection connection)
    {
        Console.WriteLine("Callback method called for "
                          + connection.name);
    }
```

```
    public static void Main()
    {
        DBManager dbMgr = new DBManager();
        dbMgr.AddConnections();

    DBManager.EnumConnectionsCallback myCallback =
    new DBManager.EnumConnectionsCallback(ActiveConnectionsCallback);

    DBManager.EnumConnections(myCallback);
    }
}
```

אם תהדר ותבצע את היישום תקבל את הפלט הבא:

```
Callback method called for DBConnection 1
Callback method called for DBConnection 2
Callback method called for DBConnection 3
Callback method called for DBConnection 4
Callback method called for DBConnection 5
```

# הגדרת נציגים כחברים סטטיים

מכיון שהלקוח צריך לממש את הנציג בכל פעם שעומדים להשתמש בו, #C מאפשרת להגדיר את השיטה שתשמש ביצירת הנציג כחברת מחלקה סטטית. להלן הדוגמה מסעיף קודם לאחר השינויים בהתאם לפורמט זה. שים לב שכעת הנציג מוגדר כחבר סטטי בשם myCallback של המחלקה. כמו כן, שים לב שניתן להשתמש בחבר זה בשיטה Main כך שהלקוח אינו נדרש לממש את הנציג.

```
using System;

class DBConnection
{
   public DBConnection(string name)
   {
      this.name = name;
   }

   protected string Name;
   public string name
   {
      get
      {
         return this.Name;
      }
```

```
        set
        {
            this.Name = value;
        }
    }
}

class DBManager
{
    static DBConnection[] activeConnections;
    public void AddConnections()
    {
        activeConnections = new DBConnection[5];
        for (int i = 0; i < 5; i++)
        {
            activeConnections[i] = new
DBConnection("DBConnection " + (i + 1));
        }
    }

    public delegate void EnumConnectionsCallback(DBConnection connection);
    public static void EnumConnections(EnumConnectionsCallback callback)
    {
        foreach (DBConnection connection in activeConnections)
        {
            callback(connection);
        }
    }
}

class Delegate2App
{
    public static DBManager.EnumConnectionsCallback myCallback =
        new DBManager.EnumConnectionsCallback(ActiveConnectionsCallback);
    public static void ActiveConnectionsCallback(DBConnection connection)
    {
        Console.WriteLine ("Callback method called for " +
                           connection.name);
    }

    public static void Main()
    {
        DBManager dbMgr = new DBManager();
        dbMgr.AddConnections();

        DBManager.EnumConnections(myCallback);
    }
}
```

הערה

מכיון שמוסכמת הכינוי התקנית עבור נציגים היא להוסיף את המילה Callback לשם השיטה שלוקחת את הנציג כארגומנט, קל מאוד לטעות ולהשתמש בשם השיטה במקום בשם הנציג. במקרה זה תקבל שגיאת הידור מבלבלת במידת מה, האומרת שציינת שיטה במקום שבו המהדר מצפה למחלקה. אם אתה מקבל שגיאה זו, זכור שהבעיה האמיתית היא שציינת שיטה במקום נציג.

# יצירת נציגים על פי צורך

בשתי הדוגמאות שראינו עד כה, הנציג נוצר בין אם אי פעם ישתמשו בו ובין אם לאו. בדוגמאות אלו ידענו שהנציג תמיד נקרא ולכן לא היתה בעיה להגדירו כך. עם זאת, כאשר מגדירים נציגים (delegates) חשוב לשקול מתי ליצור אותם. לשם דוגמה, נניח שיצירת נציג מסוים דורשת זמן ניכר ולא רוצים שתהליך זה יקרה ללא שליטה. במקרים כאלה בהם יודעים שהלקוח בדרך כלל לא יקרא לשיטת משוב נתונה, ניתן לדחות את יצירת הנציג עד לרגע שבו הוא נחוץ בפועל על ידי איחוד יצירת מופעו בתוך מאפיין. בכדי להמחיש זאת, נשנה את המחלקה DBManager כך שתשתמש במאפיין לקריאה בלבד - מאחר ונוכחת רק שיטה קוראת (getter) - ליצירת מופע של הנציג. הנציג לא ייווצר עד שתהיה התייחסות למאפיין זה.

```
using System;

class DBConnection
{
   public DBConnection(string name)
   {
      this.name = name;
   }

   protected string Name;
   public string name
   {
      get
      {
         return this.Name;
      }
      set

      {
         this.Name = value;
      }
   }
}
```

```
class DBManager
{
   static DBConnection[] activeConnections;
   public void AddConnections()
   {
      activeConnections = new DBConnection[5];
      for (int i = 0; i < 5; i++)
      {
         activeConnections[i] = new
            DBConnection("DBConnection " + (i + 1));
      }
   }

   public delegate void EnumConnectionsCallback(DBConnection connection);
   public static void EnumConnections(EnumConnectionsCallback callback)
   {
      foreach (DBConnection connection in activeConnections)
      {
         callback(connection);
      }
   }
}

class Delegate3App
{
   public DBManager.EnumConnectionsCallback myCallback
   {
      get
      {
         return new DBManager.EnumConnectionsCallback
                        (ActiveConnectionsCallback);
      }
   }

   public static void ActiveConnectionsCallback(DBConnection connection)
   {
      Console.WriteLine
         ("Callback method called for " + connection.name);
   }
```

```
    public static void Main()
    {
        Delegate3App app = new Delegate3App();

        DBManager dbMgr = new DBManager();
        dbMgr.AddConnections();

        DBManager.EnumConnections(app.myCallback);
    }
}
```

# הרכבת נציגים

היכולת להרכיב נציגים - יצירת נציג אחד מתוך נציגים מרובים - היא אחת מאותן תכונות שבהתחלה אינן נראות שימושיות במיוחד, אך כאשר תזדקק לה ודאי תהיה מאושר מכך שצוות התכנון של C# חשב עליה. הבה נבחן מספר דוגמאות בהם הרכבת נציגים היא שימושית. בדוגמה הראשונה קיימת מערכת הפצה ומחלקה המבצעת איטרציה על הפריטים עבור מיקום מסוים, כשהיא קוראת לשיטת משוב עבור כל פריט שהוא בעל ערך של "כמות במלאי" קטן מ- 50. בדוגמה מציאותית יותר של מערכת הפצה, הנוסחה מתחשבת לא רק בערך של "כמות במלאי" אלא גם ב"כמות מוזמנת" וגם "כמות בשינוע" ביחס לזמנים המובילים, והיא מקטינה את רמת מלאי הביטחון, וכן הלאה. כאן נעשה זאת באופן פשוט: אם ערך "כמות במלאי" של פריט הוא קטן מ- 50, אזי התרחש חריג.

הבעיה היא שאנו רוצים לקרוא לשתי שיטות שונות אם פריט מסוים נמצא מתחת לרמת המלאי המינימלית: אנו רוצים לרשום את האירוע, ולאחר מכן לשלוח דואר אלקטרוני למנהל הרכש. הבה נראה איך יוצרים בצורה תכנותית נציג מורכב יחיד מנציגים מרובים:

```
using System;
using System.Threading;

class Part
{
    public Part(string sku)
    {
        this.Sku = sku;

        Random r = new Random(DateTime.Now.Millisecond);
        double d = r.NextDouble() * 100;

        this.OnHand = (int)d;
    }
```

```
    protected string Sku;
    public string sku
    {
        get
        {
            return this.Sku;
        }
        set
        {
            this.Sku = value;
        }
    }

    protected int OnHand;
    public int onhand
    {
        get
        {
            return this.OnHand;
        }
        set
        {
            this.OnHand = value;
        }
    }
}

class InventoryManager
{
    protected const int MIN_ONHAND = 50;

    public Part[] parts;
    public InventoryManager()
    {
        parts = new Part[5];
        for (int i = 0; i < 5; i++)
        {
            Part part = new Part("Part " + (i + 1));

            Thread.Sleep(10); // Randomizer is seeded by time.

            parts[i] = part;
            Console.WriteLine("Adding part '{0}' on-hand = {1}",
                              part.sku, part.onhand);
        }
    }
```

```
    public delegate void OutOfStockExceptionMethod(Part part);
    public void ProcessInventory(OutOfStockExceptionMethod exception)
    {
        Console.WriteLine("\nProcessing inventory...");
        foreach (Part part in parts)
        {
            if (part.onhand < MIN_ONHAND)
            {
                Console.WriteLine
                    ("{0} ({1}) is below minimum on-hand {2}",
                    part.sku, part.onhand, MIN_ONHAND);

                exception(part);
            }
        }
    }
}

class CompositeDelegate1App
{
    public static void LogEvent(Part part)
    {
        Console.WriteLine("\tlogging event...");
    }

    public static void EmailPurchasingMgr(Part part)
    {
        Console.WriteLine("\temailing Purchasing manager...");
    }

    public static void Main()
    {
        InventoryManager mgr = new InventoryManager();

        InventoryManager.OutOfStockExceptionMethod LogEventCallback =
            new InventoryManager.OutOfStockExceptionMethod(LogEvent);

        InventoryManager.OutOfStockExceptionMethod
            EmailPurchasingMgrCallback = new
            InventoryManager.OutOfStockExceptionMethod(EmailPurchasingMgr);

        InventoryManager.OutOfStockExceptionMethod
            OnHandExceptionEventsCallback =
            EmailPurchasingMgrCallback + LogEventCallback;

        mgr.ProcessInventory(OnHandExceptionEventsCallback);
    }
}
```

ביצוע היישום מפיק תוצאות דומות לאלה:

```
Adding part 'Part 1' on-hand = 16
Adding part 'Part 2' on-hand = 98
Adding part 'Part 3' on-hand = 65
Adding part 'Part 4' on-hand = 22
Adding part 'Part 5' on-hand = 70

Processing inventory...
Part 1 (16) is below minimum on-hand 50
   logging event...
   emailing Purchasing manager...
Part 4 (22) is below minimum on-hand 50
   logging event...
   emailing Purchasing manager...
```

מכאן, השימוש בתכונה זו של השפה מאפשרת לנו להבחין באופן דינאמי אילו שיטות מהוות שיטת משוב, ליצור נציג אחד מחיבור שיטות אלו, ולהעביר את הנציג המורכב כאילו היה נציג יחיד. זמן הריצה ידאג באופן אוטומטי לכך שכל השיטות ייקראו ברצף. בנוסף לכך, ניתן גם להסיר נציגים מן הנציג המורכב באמצעות האופרטור מינוס.

יחד עם זאת, העובדה ששיטות אלו נקראות בזו אחר זו מעוררת שאלה אחת חשובה: מדוע לא פשוט לשרשר את השיטות יחד כך שכל שיטה קוראת לשיטה הבאה אחריה? בדוגמה הקודמת נוכל לעשות זאת מאחר וקיימות בה רק שתי שיטות ושתיהן נקראות תמיד יחד. אולם כעת נהפוך את הדוגמה למורכבת יותר. נניח שיש מספר חנויות במיקומים שונים כאשר כל מיקום מכתיב אילו שיטות נקראות. לדוגמה, מיקום אחד (Location1) עשוי להיות המחסן, כך שנרצה לרשום את האירוע ולשלוח דואר למנהל הרכש, ואילו עבור כל שאר המיקומים נרצה לרשום את האירוע ולשלוח דואר למנהל של אותה חנות כאשר ערך "כמות המלאי" של פריט הוא מתחת למינימום.

נוכל לעמוד בקלות בדרישות אלו על ידי יצירת נציג מורכב באופן דינאמי על פי המיקום. ללא נציגים היינו צריכים לכתוב שיטה אשר לא רק נדרשת לקבוע לאילו שיטות יש לקרוא, אלא גם לעקוב אילו שיטות כבר נקראו ולאילו שיטות עדיין יש לקרוא במהלך רצף הקריאות. כפי שתראה בקוד הבא, הודות לשימוש בנציגים הופכת פעולה מסובכת (באופן פוטנציאלי) זו לפשוטה.

```
using System;

class Part
{
   public Part(string sku)
```

```
    {
        this.Sku = sku;

        Random r = new Random(DateTime.Now.Millisecond);
        double d = r.NextDouble() * 100;

        this.OnHand = (int)d;
    }

    protected string Sku;
    public string sku
    {
        get
        {
            return this.Sku;
        }
        set
        {
            this.Sku = value;
        }
    }

    protected int OnHand;
    public int onhand
    {
        get
        {
            return this.OnHand;
        }
        set
        {
            this.OnHand = value;
        }
    }
}

class InventoryManager
{
    protected const int MIN_ONHAND = 50;
```

```
    public Part[] parts;
    public InventoryManager()
    {
        parts = new Part[5];
        for (int i = 0; i < 5; i++)
        {
            Part part = new Part("Part " + (i + 1));
            parts[i] = part;
            Console.WriteLine
                ("Adding part '{0}' on-hand = {1}",
                 part.sku, part.onhand);
        }
    }

    public delegate void OutOfStockExceptionMethod(Part part);
    public void ProcessInventory(OutOfStockExceptionMethod exception)
    {
        Console.WriteLine("\nProcessing inventory...");
        foreach (Part part in parts)
        {
            if (part.onhand < MIN_ONHAND)
            {
                Console.WriteLine
                    ("{0} ({1}) is below minimum onhand {2}",
                    part.sku, part.onhand, MIN_ONHAND);

                exception(part);
            }
        }
    }
}

class CompositeDelegate2App
{
    public static void LogEvent(Part part)
    {
        Console.WriteLine("\tlogging event...");
    }

    public static void EmailPurchasingMgr(Part part)
    {
        Console.WriteLine("\temailing Purchasing manager...");
    }

    public static void EmailStoreMgr(Part part)
    {
        Console.WriteLine("\temailing store manager...");
    }
```

```
    public static void Main()
    {
        InventoryManager mgr = new InventoryManager();
        InventoryManager.OutOfStockExceptionMethod[] exceptionMethods
          = new InventoryManager.OutOfStockExceptionMethod[3];

        exceptionMethods[0] = new
          InventoryManager.OutOfStockExceptionMethod
            (LogEvent);
        exceptionMethods[1] = new
          InventoryManager.OutOfStockExceptionMethod
            (EmailPurchasingMgr);
        exceptionMethods[2] = new
          InventoryManager.OutOfStockExceptionMethod
            (EmailStoreMgr);

        int location = 1;

        InventoryManager.OutOfStockExceptionMethod compositeDelegate;

        if (location == 2)
        {
            compositeDelegate =
                exceptionMethods[0] + exceptionMethods[1];
        }
        else
        {
            compositeDelegate =
                exceptionMethods[0] + exceptionMethods[2];
        }

        mgr.ProcessInventory(compositeDelegate);
    }
}
```

כעת ביצוע היישום לאחר הידורו יניב תוצאות שונות על סמך הערך שאתה מקצה למשתנה location.

# הגדרת אירועים באמצעות נציגים

כמעט כל היישומים עבור Windows צריכים לבצע עיבוד אירועים אסינכרוני מסוג כלשהו. חלק מאירועים אלה הם כלליים, כגון הודעות ששולחת Windows לתור ההודעות של היישום כאשר המשתמש מבצע אינטראקציה עם היישום בדרך כלשהי. חלק מהאירועים ייחודיים יותר למרחב הבעיה, כמו הדפסת חשבונית כאשר מעדכנים הזמנה.

אירועים ב-#C מצייתים לדפוס התכנון של מנוי-פרסום (publish-subscribe) שבו מחלקה מפרסמת אירוע שהיא יכולה לעורר ומחלקות אחרות מכל מספר שהוא יכולות להירשם - להיות מנויות - לאירוע זה. מרגע שאירוע מתרחש, זמן הריצה דואג ליידע כל מנוי שהאירוע התרחש.

השיטה הנקראת כתוצאה מאירוע שהתרחש מוגדרת באמצעות נציג. עם זאת, זכור מספר חוקים נוקשים ביחס לנציג הממומש בדרך זו. ראשית, הנציג חייב להיות מוגדר כלוקח שני ארגומנטים. שנית, ארגומנטים אלה מייצגים תמיד שני אובייקטים: האובייקט שעורר את האירוע (המפרסם) ואובייקט המכיל מידע על האירוע. בנוסף לכך, האובייקט השני חייב להיגזר מהמחלקה EventArgs של מסגרת NET.

נניח שרוצים לעקוב אחר שינויים ברמות המלאי. נוכל ליצור מחלקה בשם InventoryManager שתשמש תמיד לעדכון המלאי. מחלקה זו תפרסם אירוע שיתרחש בכל פעם שהמלאי ישתנה באמצעות פעולות כמו קבלת פריטים למלאי, מכירות ועדכוני מלאי פיסיים. לאחר מכן, כל מחלקה שצריכה להיות מעודכנת תירשם לאירוע זה. כך זה מקודד ב-#C על ידי שימוש בנציגים ואירועים:

```
using System;

class InventoryChangeEventArgs : EventArgs
{
   public InventoryChangeEventArgs(string sku, int change)
   {
      this.sku = sku;
      this.change = change;
   }

   string sku;
   public string Sku
   {
      get
      {
         return sku;
      }
   }

   int change;
   public int Change
   {
      get
      {
         return change;
      }
   }
}
```

```
class InventoryManager // Publisher.
{
   public delegate void InventoryChangeEventHandler
      (object source, InventoryChangeEventArgs e);
   public event InventoryChangeEventHandler OnInventoryChangeHandler;

   public void UpdateInventory(string sku, int change)
   {
      if (0 == change)
         return; // No update on null change.

      // Code to update database would go here.

      InventoryChangeEventArgs e = new
         InventoryChangeEventArgs(sku, change);

      if (OnInventoryChangeHandler != null)
         OnInventoryChangeHandler(this, e);
   }
}

class InventoryWatcher // Subscriber.
{
   public InventoryWatcher(InventoryManager inventoryManager)
   {
      this.inventoryManager = inventoryManager;
      inventoryManager.OnInventoryChangeHandler += new
InventoryManager.InventoryChangeEventHandler(OnInventoryChange);
   }
   void OnInventoryChange(object source, InventoryChangeEventArgs e)
   {
      int change = e.Change;
      Console.WriteLine("Part '{0}' was {1} by {2} units",
         e.Sku,
         change > 0 ? "increased" : "decreased",
         Math.Abs(e.Change));
   }
   InventoryManager inventoryManager;
}
```

```
class Events1App
{
   public static void Main()
   {
      InventoryManager inventoryManager =
         new InventoryManager();

      InventoryWatcher inventoryWatch =
         new InventoryWatcher(inventoryManager);

      inventoryManager.UpdateInventory("111 006 116", -2);
      inventoryManager.UpdateInventory("111 005 383", 5);
   }
}
```

הבה נסתכל על שני החברים הראשונים של המחלקה InventoryManager:

```
public delegate void InventoryChangeEventHandler
                        (object source, InventoryChangeEventArgs e);
public event InventoryChangeEventHandler OnInventoryChangeHandler;
```

שורת הקוד הראשונה היא נציג אשר כידוע מהווה הגדרה עבור חתימת שיטה. כפי שהוזכר מקודם, כל הנציגים המשמשים באירועים חייבים להיות מוגדרים כלוקחים שני ארגומנטים: אובייקט מפרסם (במקרה זה, source) ואובייקט מידע אירוע (אובייקט הנגזר מ-EventArgs). שורת הקוד השנייה משתמשת במילת המפתח event, המציינת סוג חבר שאיתו מגדירים את הנציג ואת השיטה (או השיטות) שייקראו כאשר מתרחש האירוע.

השיטה האחרונה במחלקה InventoryManager היא UpdateInventory, הנקראת בכל פעם שהמלאי משתנה. כפי שרואים, שיטה זו יוצרת אובייקט מסוג InventoryChangeEventArgs. אובייקט זה מועבר לכל המנויים ומשמש לתיאור האירוע שהתרחש.

כעת נסתכל על שתי שורות הקוד בהמשך:

```
if (OnInventoryChangeHandler != null)
        OnInventoryChangeHandler(this, e);
```

משפט התנאי if בודק אם לאירוע יש מנויים המשויכים לשיטה OnInventoryChangeHandler. אם כן - כלומר, OnInventoryChangeHandler אינה null - מעוררים את האירוע. למעשה זה כל מה שנדרש בצד המפרסם. כעת נבחן את קוד המנוי.

המנוי במקרה זה הוא מחלקה בשם InventoryWatcher. כל שעליה לבצע הוא שתי משימות פשוטות. ראשית, היא רושמת את עצמה כמנויה באמצעות מימוש נציג חדש מסוג InventoryManager.InventoryChangeEventHandler ומוסיפה נציג זה לאירוע InventoryManager.OnInventoryChangeHandler.

שים לב במיוחד לתחביר המשתמש באופרטור ההצבה המורכב += להוספת המחלקה לרשימת המנויים כך שלא יימחקו מנויים קודמים.

```
inventoryManager.OnInventoryChangeHandler
+= new InventoryManager.InventoryChangeEventHandler(OnInventoryChange);
```

הארגומנט היחיד שיש לספק כאן הוא שם השיטה הנקראת אם וכאשר יתרחש האירוע.

המשימה השנייה שהמנוי צריך לבצע היא ליישם מטפל באירוע משלו. במקרה זה, המטפל באירוע הוא InventoryWatcher.OnInventoryChange, המדפיס הודעה שמציינת את מספר הפריט ואת השינוי במלאי.

לסיום, הקוד שמבצע את היישום יוצר מופעים של המחלקות InventoryManager ו-InventoryWatcher ובכל פעם שהשיטה InventoryManager.UpdateInventory נקראת, מתרחש באופן אוטומטי אירוע הגורם לקריאה לשיטה – InventoryWatcher.OnInventoryChanged.

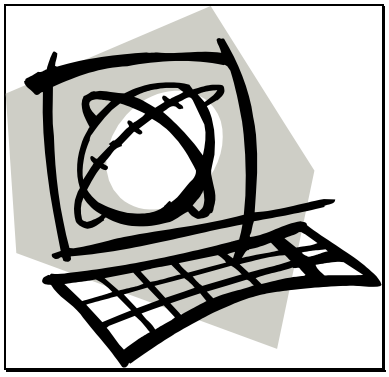
# סיכום

נציגים ב-#C הם אובייקטים מבטיחי-סוג, מאובטחים ומנוהלים, המשרתים אותה מטרה כמו של מצביעי פונקציות ב- ++C. נציגים נבדלים ממחלקות וממשקים בכך שהם מתייחסים לשיטות יחידות ומוגדרים בזמן ריצה. הם משמשים בדרך כלל לביצוע עיבודים אסינכרוניים ולהשתלת קוד מותאם לתוך נתיב הקוד של מחלקה. נציגים יכולים לשמש למספר מטרות כלליות, כולל שימוש כשיטות משוב, הגדרת שיטות סטטיות ושימוש להגדרת אירועים.

## חלק 4

# נושאים מתקדמים

# ב- C#



פרק 15

# תכנות עם ריבוי מטלות - Multithreaded Programming

מבחינה טכנית, מטלות (threads) אינן ייחודיות לשפת #C. מסיבה זו רוב הספרים העוסקים ב-#C נוטים להתרחק מנושא זה. למרות שהשתדלתי להתמקד בהקשר הייחודי ל-#C, ריבוי מטלות הוא מסוג הנושאים שרוב המתכנתים אמורים להיות בקיאים בו כאשר הם לומדים שפה חדשה זו. איני יכול בוודאות לסקור בפרק אחד את כל מרחב הסוגיות הנוגעות למטלות. עם זאת, בפרק זה אסקור את היסודות וגם מספר סוגיות בינוניות עד מתקדמות בנוגע לנטישה (aborting), תזמון (scheduling) וניהול משך החיים של מטלות. כמו כן אדון בסינכרון מטלות באמצעות המחלקות System.Monitor ו-System.Mutex והמשפט lock של #C.

## יסודות השימוש במטלות

השימוש ביישום עם מטלה בודדת הוא כמו לערוך קניות במרכול עם קופאי אחד. הפעלת קופאי אחד בלבד יקרה פחות עבור בעל המרכול והקופאי יכול לטפל ברמה נמוכה יחסית של תנועת לקוחות. אולם כאשר המרכול מכריז על יום של הנחות גדולות, התור לקופה נעשה ארוך יותר ויותר ומספר לקוחות יהיו מאד לא מאושרים עקב כך. כאן זהו התסריט הרגיל של צוואר בקבוק: הרבה מאד נתונים הזורמים לתוך מסלול צר מאד. הפתרון הוא כמובן בהגדלת מספר הקופות.

כך קורה גם כשמדובר ביישומים ומטלות. ה**שימוש בריבוי מטלות** (multithreading) מאפשר ליישום לפצל משימות כך שיפעלו באופן בלתי תלוי אחת באחרת לניצול מירבי של המעבד וזמנו של המשתמש.

עם זאת, אם תכנות עם ריבוי מטלות חדש עבורך עלי להזהירך: **השימוש במטלות אינו הבחירה הנכונה עבור כל היישומים ולעיתים יכול גם להאט את היישום**. לכן, לצד לימוד התחביר של שימוש במטלות חשוב ביותר גם להבין מתי להשתמש בו. לשם כך כללתי בסוף הפרק את הסעיף "קווים מנחים לשימוש במטלות" שיעזור לך להחליט מתי ראוי ליצור מטלות מרובות ביישום שלך. לעת עתה הבה נראה איך שימוש במטלות מיושם במסגרת NET של Microsoft.

## מטלות וריבוי משימות

**מטלה** (thread) היא יחידת עיבוד, **וריבוי משימות** (multitasking) הוא הביצוע הסימולטני של מטלות מרובות. ריבוי משימות יכול להיות בשיתופיות (cooperative) או ב- preemptive (זכות קדימה). גרסאות מוקדמות מאד של Microsoft Windows תמכו בריבוי משימות שיתופי (cooperative), כלומר כל מטלה היתה אחראית לשחרר את הבקרה כך שהמעבד יוכל לשרת מטלות אחרות. לכל אחד עם רקע במערכות הפעלה אחרות, יש סיפור משלו על היום שבו "תקע" את המחשב כי לא הציב בקוד קריאה ל-PeekMessage בכדי לאפשר למעבד לשרת מטלות אחרות במערכת. "אני צריך לעשות מה?!" היתה התגובה הטיפוסית.

עם זאת, Microsoft Windows NT - ומאוחר יותר גם Windows 95, Windows 98 ו- Windows 2000 - תומכת באותו ריבוי משימות preemptive. בריבוי משימות preemptive המעבד אחראי למתן מידה מסוימת של זמן עיבוד לכל מטלה - מה שנקרא פרוסת או **מנת זמן** (timeslice). אז מחליף המעבד בין המטלות השונות כשהוא מעניק לכל מטלה את מנת הזמן שלה. התוכנית אינה צריכה לדאוג אודות, איך ומתי מטלה צריכה לנטוש את הבקרה כדי שמטלות אחרות יבוצעו. כאן אתמקד בריבוי משימות preemptive מאחר ו-NET פועלת רק על מערכות הפעלה התומכות בכך.

הערה

בתור משתמשים אנו מכירים את המצב בו יישום 16 bit, הפועל במערכת 32 bit, לא משחרר משאבים. מצב זה "תוקע" את כל המערכת ולא נותר אלא ללחוץ Reset.

אגב, גם עם ריבוי משימות preemptive, אם מבצעים תוכנית על מכונה בעלת מעבד יחיד, למעשה אין מצב שבו מטלות מרובות מתבצעות בו-זמנית. מאחר והמעבד מחליף בין תהליכים במרווחים של מילי-שניות, רק "חשים" כאילו המטלות מתבצעות באותו הזמן. אם רוצים ביצוע מטלות מרובות בו-זמנית אמיתי צריך לפתח ולבצע קוד על מכונה בעלת יותר ממעבד אחד.

## context switching

**החלפת הקשר** (context switching) היא חלק אינטגרלי של שימוש במטלות ומושג לא פשוט עבור חלק מאיתנו. כאן אתן סקירה קצרה על מושג זה.

המעבד משתמש במונה זמן של החומרה בכדי לקבוע מתי מסתיימת מנת זמן של מטלה נתונה. כאשר מונה זה מאותת את מנת הזמן (timeslice), המעבד שומר את כל האוגרים (registers) עבור מטלה זו בתוך מחסנית.

לאחר מכן הוא מעביר את האוגרים מהמחסנית לתוך מבנה נתונים הנקרא מבנה CONTEXT. כאשר המעבד רוצה לחזור למטלה שבוצעה מקודם, הוא מבצע תהליך הפוך ומשחזר את האוגרים מתוך מבנה CONTEXT המשויך לאותה מטלה. תהליך זה כולו נקרא החלפת הקשר (context switching).

# יישום #C עם ריבוי מטלות

לפני שנבחן כמה מהדרכים השונות בהן ניתן להשתמש במטלות ב-#C, הבה נראה עד כמה קל ליצור מטלה שנייה בשפה זו. אנו נסתכל מקרוב על מרחב השמות System.Threading ובמיוחד על המחלקה Thread. בדוגמה זו אני יוצר מטלה שנייה מתוך השיטה Main. השיטה המשויכת למטלה השנייה מדפיסה לאחר מכן הודעה האומרת שהמטלה נקראה.

```
using System;
using System.Threading;

class SimpleThreadApp
{
   public static void WorkerThreadMethod()
   {
      Console.WriteLine("Worker thread started");
   }

   public static void Main()
   {
      ThreadStart worker = new ThreadStart(WorkerThreadMethod);

      Console.WriteLine("Main - Creating worker thread");

      Thread t = new Thread(worker);
      t.Start();

   Console.WriteLine("Main - Have requested the start of worker thread");
   }
}
```

אם תהדר ותבצע את היישום, תבחין שההודעה מהשיטה Main מודפסת לפני ההודעה מהמטלה worker, וזאת ההוכחה לכך שהמטלה worker אכן פועלת בצורה אסינכרונית. הבה ננתח מה קרה כאן.

הפריט הראשון שיש לציין הוא הכללת המשפט using עבור מרחב השמות System.Threading. בקרוב נכיר מרחב שמות זה. לעת עתה, מספיק להבין שמרחב שמות זה מכיל את המחלקות השונות הנחוצות למימוש ריבוי משימות בסביבת NET. כעת נסתכל על שורת הקוד הראשונה של השיטה Main:

```
ThreadStart worker = new ThreadStart(WorkerThreadMethod);
```

בכל פעם שאתה רואה את הצורה הבאה, אתה יכול להיות בטוח ש-x הוא נציג (delegate) או - כפי שלמדת בפרק 14 - הגדרה של חתימת שיטה:

```
x varName = new x (methodName);
```

לפיכך אנו יודעים ש-ThreadStart הוא נציג, אולם אין זה נציג כמו האחרים. ליתר דיוק, הנציג הוא זה אשר חייבים להשתמש בו כאשר יוצרים מטלה חדשה, והוא משמש לציון השיטה של המטלה. בהמשך אני יוצר מופע מאובייקט Thread שבו הבנאי לוקח נציג ThreadStart כארגומנט יחיד:

```
Thread t = new Thread(worker);
```

לאחר מכן אני קורא לשיטה Start של האובייקט Thread וזה גורם לקריאה לשיטה WorkerThreadMethod.

זה הכל! בסך הכל נדרשו שלוש שורות קוד בכדי להגדיר ולהפעיל את המטלה ואת שיטת המטלה עצמה. כעת נבחן לעומק את מרחב השמות System.Threading ואת מחלקותיו, המאפשרות לכל זה לקרות.

# עבודה עם מטלות

כל משימות היצירה והניהול של מטלות מושגות באמצעות השימוש במחלקה Thread של System.Threading ולכן נתחיל מכאן.

## AppDomain

בסביבת NET מתבצעות מטלות בדבר הנקרא AppDomain. לעיתים אומרים ש-AppDomain הוא המקבילה לתהליך של Win32 בכך שהוא מציע רבים מאותם היתרונות, כולל היכולת להגיב בצורה נבונה לכישלון בלתי צפוי, והיכולת להיות מופעל ומופסק באופן בלתי תלוי. אמנם זו השוואה טובה, אך היא אינה תופסת בנוגע למטלות. ב- Win32 מטלה מסויגת לתהליך בודד, כפי שתיארתי מקודם בהחלפת הקשר. מטלה בתהליך אחד אינה יכולה להפעיל שיטה במטלה השייכת לתהליך אחר. עם זאת, ב-NET מטלות יכולות לחצות את גבולות ה-AppDomain, ושיטה במטלה אחת יכולה לקרוא לשיטה ב-AppDomain אחר. לפיכך, הנה הגדרה טובה יותר של AppDomain: תהליך לוגי בתוך תהליך פיסי.

# המחלקה Thread

המחלקה Thread משמשת עבור רוב המשימות הקשורות במטלות. סעיף זה בוחן את השימוש במחלקה זו לביצוע המשימות הבסיסיות.

## יצירת מטלות ואובייקטים של Thread

ניתן ליצור מופע מאובייקט Thread בשתי דרכים. דרך אחת כבר ראינו: ליצור מטלה חדשה ולקבל בתהליך זה אובייקט Thread אשר באמצעותו מתַפְעלים את המטלה החדשה. הדרך האחרת לקבל אובייקט Thread היא באמצעות קריאה לשיטה הסטטית Thread.CurrentThread עבור המטלה המתבצעת כרגע.

## ניהול אורך החיים של מטלות

קיימות משימות רבות שצריך אולי לבצע כדי לנהל את פעילות או את חיי המטלה. ניתן לשלוט בכל המשימות הללו באמצעות שיטות Thread השונות. לדוגמה, קיים צורך די שכיח להשהות מטלה לפרק זמן נתון. לשם כך ניתן לקרוא לשיטה Thread.Sleep. שיטה זו לוקחת ארגומנט יחיד המייצג את משך הזמן (במילי-שניות) הרצוי להשהיית המטלה. שים לב ששיטה זו היא סטטית ולא ניתן לקרוא לה עם מופע של אובייקט Thread. יש סיבה טובה מאוד לכך, אינך רשאי לקרוא לשיטה זו על כל מטלה אחרת פרט לזו המתבצעת כרגע. השיטה הסטטית Thread.Sleep קוראת לשיטה הסטטית CurrentThread, אשר לאחר מכן משהה את המטלה למשך הזמן המוגדר. הנה דוגמה:

```
using System;
using System.Threading;

class ThreadSleepApp
{
   public static void WorkerThreadMethod()
   {
      Console.WriteLine("Worker thread started");

      int sleepTime = 5000;

      Console.WriteLine("\tsleeping for {0} seconds", sleepTime / 1000);
      Thread.Sleep(sleepTime); // Sleep for five seconds.
      Console.WriteLine("\twaking up");
   }
```

```
    public static void Main()
    {
        ThreadStart worker = new ThreadStart(WorkerThreadMethod);

        Console.WriteLine("Main - Creating worker thread");

        Thread t = new Thread(worker);
        t.Start();

        Console.WriteLine
            ("Main - Have requested the start of worker thread");
    }
}
```

קיימות מספר דרכים נוספות לקרוא לשיטה Thread.Sleep. ראשית, קריאה ל-Thread.Sleep עם הערך 0, גורמת למטלה הנוכחית לוותר על החלק הלא מנוצל במנת הזמן שלה. אם מספקים ערך של Timeout.Infinite גורמים למטלה להשהות את עצמה ללא גבול עד אשר ההשהיה מופסקת על ידי מטלה אחרת הקוראת לשיטה Thread.Interrupt של המטלה המושהית.

הדרך השנייה להשהות ביצוע של מטלה היא באמצעות השיטה Thread.Suspend. קיימים מספר הבדלים עיקריים בין שתי הטכניקות. ראשית, ניתן לקרוא לשיטה Thread.Suspend על המטלה המתבצעת כרגע או על מטלה אחרת. שנית, מרגע שמטלה מושהית בדרך זו, רק מטלה אחרת יכולה לגרום להמשך ביצועה באמצעות השיטה Thread.Resume. שים לב כי מרגע שמטלה משהה מטלה אחרת, המטלה המשהה אינה נחסמת והקריאה חוזרת מייד. כמו כן, אין חשיבות למספר הפעמים שהשיטה Thread.Suspend נקראת עבור מטלה נתונה, קריאה אחת ל-Thread.Resume גורמת למטלה לחדש את ביצועה.

## חיסול מטלות

אם מתעורר הצורך לחסל מטלה, ניתן לבצע זאת בעזרת קריאה לשיטה Thread.Abort. תוכל לכתוב cache handler כדי לתפוס את החריג ThreadAbortException. הקוד הבא אינו נחוץ לחיסול מטלה, אבל כדאי שתבין את דרך פעולתו. השיטה Main משהה את ביצועה למשך שתי שניות בכדי להבטיח שזמן הריצה הפעיל את המטלה worker. מרגע הפעלתה, מטלה זו מתחילה למנות עד עשר עם השהיה של חצי שנייה בין כל מספר. כאשר השיטה Main מחדשת את הביצוע לאחר ההשהיה בת שתי השניות, היא גורמת לנטישה של המטלה Worker. שים לב שהבלוק finally מתבצע לאחר הנטישה.

```
using System;
using System.Threading;

class ThreadAbortApp
{
    public static void WorkerThreadMethod()
    {
        try
        {
            Console.WriteLine("Worker thread started");

            Console.WriteLine
                ("Worker thread - counting slowly to 10");
            for (int i = 0; i < 10; i++)
            {
                Thread.Sleep(500);
                Console.Write("{0}...", i);
            }

            Console.WriteLine("Worker thread finished");
        }
        catch(ThreadAbortException e)
        {
        }
        finally
        {
            Console.WriteLine
                ("Worker thread -
                 I can't catch the exception, but I can cleanup");
        }
    }

    public static void Main()
    {
        ThreadStart worker = new ThreadStart(WorkerThreadMethod);

        Console.WriteLine("Main - Creating worker thread");

        Thread t = new Thread(worker);
        t.Start();
```

```
        // Give the worker thread time to start.
        Console.WriteLine("Main - Sleeping for 2 seconds");
        Thread.Sleep(2000);

        Console.WriteLine("\nMain - Aborting worker thread");
        t.Abort();
    }
}
```

הנה הפלט המתקבל לאחר הידור וביצוע היישום:

```
Main - Creating worker thread
Main - Sleeping for 2 seconds
Worker thread started
Worker thread - counting slowly to 10
0...1...2...3...
Main - Aborting worker thread
Worker thread - I can't catch the exception, but I can cleanup
```

יש גם להבין שכאשר השיטה Thread.Abort נקראת, המטלה אינה חדלה לפעול מייד. זמן הריצה ממתין עד אשר המטלה מגיעה למה שמתואר בתיעוד כ"נקודת בטוחה" ("safe-point"). לפיכך, אם הקוד שלך תלוי במשהו שקורה לאחר הנטישה ועליך להיות בטוח שהמטלה הפסיקה, תוכל להשתמש בשיטה Thread.Join. זוהי קריאה סינכרונית, כלומר היא אינה חוזרת עד אשר המטלה הופסקה לגמרי. לסיום, שים לב כי מרגע שאתה נוטש מטלה, היא אינה יכולה לפעול מחדש. במקרה זה, למרות שיש לך אובייקט Thread חוקי, אינך יכול לעשות בו כל שימוש מועיל במונחים של ביצוע קוד.

## תזמון מטלות

כאשר המעבד מחליף בין מטלות ברגע שמסתיימת מנת זמן נתונה של מטלה, התהליך של בחירת המטלה הבאה לביצוע רחוק מלהיות אקראי. לכל מטלה משויכת רמת קדימות האומרת למעבד איך יש לתזמן אותה ביחס למטלות אחרות במערכת. רמת קדימות של Normal נקבעת כברירת מחדל עבור כל המטלות הנוצרות בתוך סביבת הריצה. עבור מטלות הנוצרות מחוץ לסביבת הריצה נשמרת הקדימות המקורית. ניתן להגדיר ולקרוא את ערך הקדימות באמצעות המאפיין Thread.Priority. השיטה הכותבת של המאפיין Thread.Priority לוקחת ערך מסוג Thread.ThreadPriority שהוא enum המגדיר את הערכים האלה: Highest, AboveNormal, Normal, BelowNormal ו-Lowest.

בכדי להמחיש את האופן שבו קדימויות יכולות להשפיע אפילו בקוד פשוט ביותר, נסתכל על הדוגמה הבאה שבה מטלה אחת מונה מ- 1 עד 10 ומטלה שנייה מונה מ- 11 עד 20. שים לב ללולאה המקוננת בתוך כל שיטת WorkerThread. כל אחת מהלולאות נמצאת שם בכדי לייצג עבודה שהמטלה היתה מבצעת ביישום אמיתי. מאחר ושיטות אלה ממש אינן עושות כאן דבר, בלי לולאות אלו כל מטלה היתה מסיימת את עבודתה במנת הזמן הראשונה שלה!

```
using System;
using System.Threading;

class ThreadSchedule1App
{
    public static void WorkerThreadMethod1()
    {
        Console.WriteLine("Worker thread started");

        Console.WriteLine
          ("Worker thread - counting slowly from 1 to 10");
        for (int i = 1; i < 11; i++)
        {
            for (int j = 0; j < 100; j++)
            {
                Console.Write(".");
                // Code to imitate work being done.
                int a;
                a = 15;
            }
            Console.Write("{0}", i);
        }

        Console.WriteLine("Worker thread finished");
    }

    public static void WorkerThreadMethod2()
    {
        Console.WriteLine("Worker thread started");

        Console.WriteLine
          ("Worker thread - counting slowly from 11 to 20");
        for (int i = 11; i < 20; i++)
        {
            for (int j = 0; j < 100; j++)
            {
                Console.Write(".");
                // Code to imitate work being done.
                int a;
                a = 15;
            }
            Console.Write("{0}", i);
        }

        Console.WriteLine("Worker thread finished");
    }
```

```
  public static void Main()
   {
      ThreadStart worker1 = new ThreadStart(WorkerThreadMethod1);
      ThreadStart worker2 = new ThreadStart(WorkerThreadMethod2);

      Console.WriteLine("Main - Creating worker threads");

      Thread t1 = new Thread(worker1);
      Thread t2 = new Thread(worker2);

      t1.Start();
      t2.Start();
   }
}
```

אם תבצע את היישום תקבל את הפלט הבא. שים לב שהפלט קוצץ לשם קיצור - רוב הנקודות הוסרו.

```
Main - Creating worker threads
Worker thread started
Worker thread started
Worker thread - counting slowly from 1 to 10
Worker thread - counting slowly from 11 to 20
......1......11......2......12......3......13
```

כפי שרואים, שתי המטלות מקבלות זמן עיבוד שווה. כעת נשנה את המאפיין Priority עבור כל מטלה כמו בקוד הבא - למטלה הראשונה נקבע את הקדימות הגבוהה ביותר המותרת ולשנייה את הקדימות הנמוכה ביותר - ונקבל תוצאה שונה בהרבה.

```
using System;
using System.Threading;

class ThreadSchedule2App
{
   public static void WorkerThreadMethod1()
   {
      Console.WriteLine("Worker thread started");

      Console.WriteLine
         ("Worker thread - counting slowly from 1 to 10");
      for (int i = 1; i < 11; i++)
      {
         for (int j = 0; j < 100; j++)
         {
            Console.Write(".");
            // Code to imitate work being done.
            int a;
            a = 15;
         }
```

```
            Console.Write("{0}", i);
        }

        Console.WriteLine("Worker thread finished");
    }

    public static void WorkerThreadMethod2()
    {
        Console.WriteLine("Worker thread started");

        Console.WriteLine
            ("Worker thread - counting slowly from 11 to 20");
        for (int i = 11; i < 20; i++)
        {
            for (int j = 0; j < 100; j++)
            {
                Console.Write(".");
                // Code to imitate work being done.
                int a;
                a = 15;
            }
            Console.Write("{0}", i);
        }

        Console.WriteLine("Worker thread finished");
    }

    public static void Main()
    {
        ThreadStart worker1 = new ThreadStart(WorkerThreadMethod1);
        ThreadStart worker2 = new ThreadStart(WorkerThreadMethod2);

        Console.WriteLine("Main - Creating worker threads");

        Thread t1 = new Thread(worker1);
        Thread t2 = new Thread(worker2);

        t1.Priority = ThreadPriority.Highest;
        t2.Priority = ThreadPriority.Lowest;

        t1.Start();
        t2.Start();
    }
}
```

התוצאות (המקוצרות) מוצגות להלן. שים לב שלמטלה השנייה נקבעה קדימות כה נמוכה כך שהיא אינה מקבלת אפילו מחזור מעבד אחד עד אשר המטלה הראשונה מסיימת את עבודתה.

```
Main - Creating worker threads
Worker thread started
Worker thread started
Worker thread - counting slowly from 1 to 10
Worker thread - counting slowly from 11 to 20
......1......2......3......4......5......6......7......8......9......10.....
11.....12.....13.....14.....15.....16.....17.....18.....19.....20
```

זכור שכאשר אומרים למעבד מהי רמת הקדימות הרצויה עבור מטלה נתונה, בסופו של דבר מערכת ההפעלה היא זו אשר משתמשת בערך הזה כחלק מאלגוריתם התזמון שלה אותו היא מוסרת למעבד. בסביבת NET, אלגוריתם זה מבוסס על רמת הקדימות שבה השתמשנו קודם (באמצעות המאפיין Thread.Priority) ביחד עם הערכים priority class ו- dynamic boost של התהליך. כל הערכים האלה משמשים ליצירת ערך מספרי (0-31 על מעבד Intel) המייצג את קדימות המטלה. המטלה שהיא בעלת הערך הגבוה ביותר היא זו עם הקדימות הגבוהה ביותר.

הערה אחרונה בנושא תזמון מטלות: השימוש בתזמון מטלות צריך להיות זהיר. נניח שיש יישום עם ממשק משתמש גרפי ומספר מטלות worker המבצעות עבודה אסינכרונית כלשהי. אם תקבע למטלות אלו קדימות גבוהה, ממשק המשתמש עלול להתנהג באופן עצל כי המטלה הראשית שבתוכה מתבצע יישום GUI מקבלת פחות מחזורי CPU. בדרך כלל, הטוב ביותר הוא להשאיר את ערך ברירת המחדל Normal הנקבע למטלה, אלא אם יש לך סיבה מיוחדת לתזמן מטלה עם קדימות גבוהה יותר.

במצב שבו מספר מטלות מתבצעות באותה קדימות, הן כולן מקבלות זמן עיבוד שווה. תזמון זה נקרא round robin scheduling.

# בטיחות וסינכרון מטלות

כאשר מתכנתים לסביבה של מטלה אחת, בדרך כלל נכתבות השיטות כך שבמספר נקודות בקוד, האובייקט נמצא זמנית במצב לא חוקי. כמובן שאם רק מטלה אחת ניגשת לאובייקט בכל זמן שהוא, מובטח לך שכל מטלה תסיים את ביצועה לפני שמטלה אחרת תיקרא - כלומר, האובייקט נמצא תמיד במצב חוקי עבור כל אחד מלקוחותיו. עם זאת, כאשר מדובר ביותר ממטלה אחת, מגיעים בקלות למצבים בהם המעבד מחליף למטלה אחרת כאשר אובייקט נמצא במצב לא חוקי. אם אותה מטלה מנסה גם כן להשתמש באותו אובייקט, התוצאות יכולות להיות בלתי צפויות למדי. לפיכך, פירוש המונח "בטיחות מטלה" ("thread safety") הוא שחברי האובייקט שומרים תמיד על מצב חוקי כאשר מטלות מרובות משתמשות באובייקט בו-זמנית.

אם כן, איך מונעים מצב בלתי צפוי זה? כרגיל בתכנות, קיימות מספר דרכים לטפל בבעיה ידועה זו. בחלק זה אסקור רק את האמצעי השכיח ביותר: סינכרון. באמצעות סינכרון מגדירים **מקטעים קריטיים** (critical sections) של קוד אשר רק מטלה אחת יכולה להיכנס לתוכם בכל זמן נתון, ובכך מבטיחים הסתרת מצבים לא חוקיים זמניים של האובייקט מלקוחותיו. נדון במספר אמצעים להגדרת מקטעים קריטיים, כולל המחלקות Monitor ו-Mutex של NET וגם משפט lock של #C.

## הגנה על קוד באמצעות המחלקה Monitor

המחלקה System.Monitor מאפשרת לך לרשום את הגישה לבלוקים של קוד באמצעות מנעולים ואיתותים. לדוגמה, שיטה המעדכנת מסד נתונים אשר אינה יכולה להתבצע על ידי שתיים או יותר מטלות בו-זמנית. אם העבודה המתבצעת על ידי אותה שיטה "זוללת" זמן באופן מיוחד, ויש מספר מטלות אשר עשויות כל אחת לקרוא לשיטה זו, הדבר עלול לגרום לבעיות חמורות. המחלקה Monitor באה למנוע מקרים אלה. נסתכל על דוגמת הסינכרון הבאה, שבה שתי מטלות קוראות כל אחת לשיטה Database.SaveData.

```
using System;
using System.Threading;

class Database
{
   public void SaveData(string text)
   {
      Console.WriteLine("Database.SaveData - Started");

      Console.WriteLine("Database.SaveData - Working");
      for (int i = 0; i < 100; i++)
      {
         Console.Write(text);
      }

      Console.WriteLine("\nDatabase.SaveData - Ended");
   }
}

class ThreadMonitor1App

{
   public static Database db = new Database();
```

```
    public static void WorkerThreadMethod1()
    {
        Console.WriteLine("Worker thread #1 - Started");

        Console.WriteLine
            ("Worker thread #1 -Calling Database.SaveData");
        db.SaveData("x");

        Console.WriteLine("Worker thread #1 - Returned from Output");
    }

    public static void WorkerThreadMethod2()
    {
        Console.WriteLine("Worker thread #2 - Started");

        Console.WriteLine
            ("Worker thread #2 - Calling Database.SaveData");
        db.SaveData("o");

        Console.WriteLine("Worker thread #2 - Returned from Output");
    }

    public static void Main()
    {
        ThreadStart worker1 = new ThreadStart(WorkerThreadMethod1);
        ThreadStart worker2 = new ThreadStart(WorkerThreadMethod2);

        Console.WriteLine("Main - Creating worker threads");

        Thread t1 = new Thread(worker1);
        Thread t2 = new Thread(worker2);

        t1.Start();
        t2.Start();
    }
}
```

כאשר אתה מהדר ומריץ יישום זה, הפלט המתקבל הוא תערובת של x ו-o, מה שמעיד על כך שהשיטה Database.SaveData מתבצעת בו-זמנית על ידי שתי המטלות (גם כאן הפלט קוצץ לשם קיצור).

```
Main - Creating worker threads

Worker thread #1 - Started
Worker thread #2 - Started

Worker thread #1 - Calling Database.SaveData
Worker thread #2 - Calling Database.SaveData

Database.SaveData - Started
Database.SaveData - Started

Database.SaveData - Working
Database.SaveData - Working
xoxoxox
```

```
        Console.WriteLine("\nDatabase.SaveData - Ended");

        Monitor.Exit(this);
    }
}

class ThreadMonitor2App
{
    public static Database db = new Database();

    public static void WorkerThreadMethod1()
    {
        Console.WriteLine("Worker thread #1 - Started");

        Console.WriteLine
          ("Worker thread #1 - Calling Database.SaveData");
        db.SaveData("x");

        Console.WriteLine("Worker thread #1 - Returned from Output");
    }

    public static void WorkerThreadMethod2()
    {
        Console.WriteLine("Worker thread #2 - Started");

        Console.WriteLine
           ("Worker thread #2 - Calling Database.SaveData");
        db.SaveData("o");

        Console.WriteLine("Worker thread #2 - Returned from Output");
    }

    public static void Main()
    {
        ThreadStart worker1 = new ThreadStart(WorkerThreadMethod1);
        ThreadStart worker2 = new ThreadStart(WorkerThreadMethod2);

        Console.WriteLine("Main - Creating worker threads");

        Thread t1 = new Thread(worker1);
        Thread t2 = new Thread(worker2);

        t1.Start();
        t2.Start();
    }
}
```

שים לב בפלט הבא, כי על אף הקריאה של המטלה השנייה לשיטה Database.SaveData, השיטה Monitor.Enter חוסמת אותה עד אשר המטלה הראשונה משחררת את המנעול.

```
Main - Creating worker threads

Worker thread #1 - Started
Worker thread #2 - Started

Worker thread #1 - Calling Database.SaveData
Worker thread #2 - Calling Database.SaveData

Database.SaveData - Started
Database.SaveData - Working
xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
Database.SaveData - Ended

Database.SaveData - Started

Worker thread #1 - Returned from Output

Database.SaveData - Working
oooooooooooooooooooooooooooooooooooooooo
Database.SaveData - Ended

Worker thread #2 - Returned from Output
```

## משפט lock

למרות שמשפט lock של #C אינו תומך בכל מערך התכונות הקיים במחלקה Thread, הוא כן מאפשר לקבל ולשחרר מנעול מוניטור (monitor lock). בכדי להשתמש במשפט lock, פשוט מגדירים את המשפט כשהקוד שיש לנעול אותו מוקף בסוגריים. הסוגריים מציינים את נקודות ההתחלה והסיום של הקוד שיש להגן עליו, כך שאין צורך במשפט unlock. הקוד הבא מייצר את אותו פלט מסונכרן כמו בדוגמאות הקודמות:

```
using System;
using System.Threading;

class Database
{
   public void SaveData(string text)
   {
      lock(this)
      {
         Console.WriteLine("Database.SaveData - Started");
```

```
            Console.WriteLine("Database.SaveData - Working");
            for (int i = 0; i < 100; i++)
            {
                Console.Write(text);
            }

            Console.WriteLine("\nDatabase.SaveData - Ended");
        }
    }
}

class ThreadLockApp
{
    public static Database db = new Database();

    public static void WorkerThreadMethod1()
    {
        Console.WriteLine("Worker thread #1 - Started");

        Console.WriteLine
            ("Worker thread #1 - Calling Database.SaveData");
        db.SaveData("x");

        Console.WriteLine("Worker thread #1 - Returned from Output");
    }

    public static void WorkerThreadMethod2()
    {
        Console.WriteLine("Worker thread #2 - Started");

        Console.WriteLine
            ("Worker thread #2 - Calling Database.SaveData");
        db.SaveData("o");

        Console.WriteLine("Worker thread #2 - Returned from Output");
    }

    public static void Main()
    {
        ThreadStart worker1 = new ThreadStart(WorkerThreadMethod1);
        ThreadStart worker2 = new ThreadStart(WorkerThreadMethod2);
```

```
        Console.WriteLine("Main - Creating worker threads");

        Thread t1 = new Thread(worker1);
        Thread t2 = new Thread(worker2);

        t1.Start();
        t2.Start();
    }
}
```

## סינכרון קוד באמצעות המחלקה Mutex

המחלקה Mutex, המוגדרת במרחב השמות System.Threading, היא ייצוג זמן ריצה של מערכת Win32 בסיסית בעלת אותו שם. ניתן להשתמש ב-mutex לרישום גישה לקוד כמו שמשתמשים במנעול מוניטור, אבל אובייקטים של mutex איטיים יותר בגלל גמישותם המוגדלת. מקורו של המונח mutex הוא הביטוי mutually exclusive, וכפי שרק מטלה אחת יכולה לקבל מנעול מוניטור עבור אובייקט נתון, כך יכולה מטלה אחת בכל זמן נתון לקבל mutex נתון.

אפשר ליצור mutex ב-#C עם אחד מהבנאים הבאים:

```
Mutex( )
Mutex(bool initiallyOwned)
Mutex(bool initiallyOwned, string mutexName)
```

הבנאי הראשון יוצר mutex ללא שם, הנרכש על ידי המטלה הנוכחית. לפיכך, mutex נעול על ידי המטלה הנוכחית. הבנאי השני מקבל דגל בוליאני המציין האם המטלה שיוצרת את mutex רוצה לרכוש אותו (לנעול אותו). הבנאי השלישי מאפשר לציין האם המטלה הנוכחית היא בעלת mutex וכן את שם ה-mutex. כעת נשלב Mutex לרישום הגישה לשיטה Database.SaveData:

```
using System;
using System.Threading;

class Database
{
    Mutex mutex = new Mutex(false);

    public void SaveData(string text)
    {
        mutex.WaitOne();

        Console.WriteLine("Database.SaveData - Started");
```

```
        Console.WriteLine("Database.SaveData - Working");
        for (int i = 0; i < 100; i++)
        {
            Console.Write(text);
        }

        Console.WriteLine("\nDatabase.SaveData - Ended");

        mutex.Close();
    }
}

class ThreadMutexApp
{
    public static Database db = new Database();

    public static void WorkerThreadMethod1()
    {
        Console.WriteLine("Worker thread #1 - Started");

        Console.WriteLine
            ("Worker thread #1 - Calling Database.SaveData");
        db.SaveData("x");

        Console.WriteLine("Worker thread #1 - Returned from Output");
    }

    public static void WorkerThreadMethod2()
    {
        Console.WriteLine("Worker thread #2 - Started");

        Console.WriteLine
            ("Worker thread #2 - Calling Database.SaveData");
        db.SaveData("o");

        Console.WriteLine("Worker thread #2 - Returned from Output");
    }

    public static void Main()
    {
        ThreadStart worker1 = new ThreadStart(WorkerThreadMethod1);
        ThreadStart worker2 = new ThreadStart(WorkerThreadMethod2);

        Console.WriteLine("Main - Creating worker threads");

        Thread t1 = new Thread(worker1);
        Thread t2 = new Thread(worker2);

        t1.Start();
        t2.Start();
    }
}
```

כעת המחלקה Database מגדירה שדה Mutex. עדיין איננו רוצים שהמטלה תרכוש את mutex כי אין לנו דרך להיכנס לתוך השיטה SaveData. השורה הראשונה של השיטה SaveData מראה איך מנסים לרכוש את mutex - באמצעות השיטה Mutex.WaitOne. בסוף השיטה יש קריאה לשיטה Close, המשחררת את mutex.

השיטה WaitOne מועמסת בכדי לספק יותר גמישות במונחים של יכולת להגדיר כמה זמן תמתין המטלה ל-mutex עד שיהיה זמין. צורות העמסה אלו הן:

```
WaitOne( )
WaitOne(TimeSpan time, bool exitContext)
WaitOne(int milliseconds, bool exitContext)
```

ההבדל הבסיסי בין העמסות אלו הוא בכך שהגירסה הראשונה - בה השתמשנו בדוגמה - תמתין ללא גבול, ואילו הגרסאות השנייה והשלישית ימתינו במשך הזמן המוגדר, המבוטא באמצעות ערך מסוג TimeSpan או ערך מסוג int.

## בטיחות מטלות ומחלקות NET

שאלה אחת שאני רואה לעיתים קרובות בקבוצות דיון ורשימות דיוור היא האם כל המחלקות System.* של NET הן בטוחות-מטלה. התשובה היא שלילית, והמחלקות גם אינן צריכות להיות כאלו. ביצועי מערכת היו נפגמים בצורה חמורה אם כל המחלקות היו רושמות את הגישה לפונקציונליות שלהן. תאר לך את הניסיון להשתמש באחת ממחלקות האוספים, אם היא היתה רוכשת מנעול מוניטור בכל פעם שאתה קורא לשיטה Add שלה. נניח שאתה יוצר מופע מאובייקט אוסף ומוסיף לו אלף אובייקטים. הביצועים היו ירודים - עד לנקודה שבה לא ניתן היה להשתמש במערכת.

# קווים מנחים לשימוש במטלות

מתי צריך להשתמש במטלות ומתי, אם בכלל, השימוש בהן מזיק ויש להימנע ממנו? בחלק זה אתאר את שני התסריטים השכיחים שבהם מטלות יכולות להיות בעלות ערך עבור יישום, וכמה מצבים בהם הדרך הטובה ביותר היא להימנע משימוש במטלות מרובות.

## מתי להשתמש במטלות

צריך להשתמש במטלות כאשר שואפים להגביר פעילות בו-זמנית, לפישוט התכנון ולניצול טוב יותר של זמן המעבד, כמתואר בסעיפים הבאים.

### פעילות בו-זמנית מוגברת

לעיתים קרובות יישומים צריכים להשלים יותר ממשימה אחת בזמן נתון. לדוגמה, פעם כתבתי מערכת לטעינת מסמכים עבור בנקים, אשר ניגשה לנתונים בתוך תקליטורים (דיסקים אופטיים) שאוחסנו בתוך שרתים של תקליטורים (jukeboxes). דמיין את כמויות הנתונים המסיביות עליהן מדובר כאן; תאר לך מתקן כזה עם כונן יחיד ו- 50 תקליטורים המספקים נתונים בנפחים של gb (gigabytes) רבים. לעיתים היו נדרשות 5 עד 10 שניות בכדי לטעון תקליטור ולמצוא את המסמך המבוקש. מיותר לציין שאין זו בדיוק ההגדרה לפרודוקטיביות אם היישום שלי היה חוסם את קלט המשתמש במהלך טעינת המסמכים. לפיכך, בכדי לטפל במקרה של קלט משתמש, הפעלתי מטלה נוספת לביצוע העבודה הפיסית של טעינת הנתונים, תוך מתן אפשרות למשתמש להמשיך בעבודה. המטלה תודיע למטלה הראשית כאשר המסמך נטען. זו דוגמה מצוינת של פעילויות בלתי תלויות - טעינת מסמך וטיפול בממשק המשתמש - הניתנות לטיפול בשתי מטלות נפרדות.

### פישוט התכנון

דרך מקובלת לפשט את התכנון של מערכות מורכבות הוא השימוש בתורים ובעיבוד אסינכרוני. בתכנון כזה מגדירים תורים המטפלים באירועים שונים המתרחשים במערכת. במקום לקרוא ישירות לשיטות, נוצרים אובייקטים המוצבים בתור כדי להיות מטופלים. בקצה האחר של התורים נמצאות תוכניות שירות עם מטלות מרובות המוגדרות "להאזין" להודעות הזורמות לתורים אלה. היתרון בסוג זה של תכנון מפושט הוא בכך שהוא מעניק למערכות יכולת הרחבה, אמינות וחסינות.

### ניצול טוב יותר של זמן המעבד

פעמים רבות קורה שיישום אינו עושה דבר בעוד שעדיין הוא נהנה ממנת הזמן שלו. בדוגמת שליפת המסמכים הקודמת, מטלה אחת המתינה לטעינת התקליטור בכונן. ברור שהמתנה זו קשורה בסוגיית חומרה ולא דורשת שימוש ב-CPU. דוגמאות אחרות לזמני המתנה הן כאשר אתה מדפיס מסמך, או ממתין לכונן הקשיח או לכונן התקליטורים. בכל מקרה, ה-CPU אינו מנוצל. מקרים כאלה הם מועמדים ראויים להעברה למטלות רקע.

## מתי אין להשתמש במטלות

שגיאה נפוצה של אלה מבינינו החדשים בתכנות מטלות, היא הניסיון לנצל אותם בכל יישום. זה יכול להיות הרבה יותר גרוע מאשר לא להשתמש בהם כלל! כמו עם כל כלי במאגר התכנות שלך, עליך להשתמש במטלות רק כאשר ראוי לעשות כן. עליך להימנע משימוש במטלות מרובות ביישומים לפחות במקרים הבאים (המתוארים בהמשך): כאשר העלויות גדולות מן התועלת, כאשר לא בחנת את הביצועים בכל אחד המקרים, וכאשר אינך מוצא סיבה מיוחדת להשתמש במטלות.

### עלויות גדולות מן התועלת

כפי שראית בסעיף "בטיחות וסינכרון מטלות" כתיבת יישומים עם ריבוי מטלות דורשת זמן ומאמץ בתכנון הקוד. יהיו זמנים שבהם הרווחים הקלושים המושגים עקב כך, אינם שווים את הזמן הנוסף הנדרש בכדי להפוך את הקוד לבטוח-מטלות.

### לא בחנת את הביצועים בכל אחד מהמקרים

אם אתה חדש בתכנות עם ריבוי מטלות, אולי תופתע לגלות שלעיתים קרובות התקורה הנדרשת ליצירת ותזמון מטלות על ידי המעבד בכדי להשיג שיפורים צנועים בניצול המעבד, מביאה לכך שיישום של מטלה אחת פועל מהר יותר! הכל תלוי במה שאתה עושה ובאם אתה באמת מפצל משימות בלתי תלויות למטלות. לדוגמה, אם אתה צריך לקרוא שלושה קבצים מדיסק, הפעלת שלוש מטלות לשם כך לא תשפר את הביצועים מאחר וכל אחת צריכה להשתמש באותו דיסק קשיח. לפיכך, הקפד תמיד לבחון אבטיפוס של גרסת מטלה בודדת מול גרסת ריבוי מטלות של המערכת שלך, לפני שאתה מכניס את עצמך להוצאות והשקעת זמן נוספת בניסיון לתכנן פתרון שעלול להביא למעשה לתוצאה הפוכה במונחים של ביצועים.

## סיכום

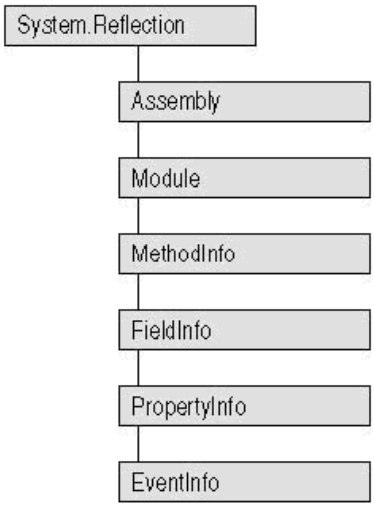
השימוש בריבוי מטלות מאפשר ליישומים לפצל משימות כך שיפעלו באופן בלתי תלוי לניצול מירבי של זמן המעבד. עם זאת, הוספת מטלות מרובות ליישום אינה הבחירה הנכונה בכל מצב ולעיתים עלולה להאט את ביצועי היישום. יצירת וניהול מטלות ב-#C נעשית באמצעות המחלקה System.Threading.Thread. מושג חשוב המתייחס ליצירה ולשימוש במטלות הוא בטיחות מטלות. בטיחות מטלות פירושה שחברי האובייקט שומרים תמיד על מצב חוקי כאשר מטלות מרובות משתמשות בהם בו-זמנית. לצד לימוד התחביר של שימוש בריבוי מטלות, חשוב גם להבין מתי להשתמש בו: להגברת בו-זמניות, לפישוט תכנון ולניצול טוב יותר של זמן מעבד.

# פרק 16 Reflection

בפרק 2 התייחסתי בקצרה לעובדה שהמהדר יוצר קובץ הפעלה נייד סטנדרטי של Win 32 (PE - portable executable) הכולל בעיקר קוד MSIL ומטא-נתונים. תכונה חזקה של NET מאפשרת לכתוב קוד גישה למטא-נתונים של יישום באמצעות תהליך הנקרא **שיקוף** (reflection). במילים פשוטות, שיקוף הוא היכולת לגלות מידע על סוגי נתונים בזמן ריצה. פרק זה מתאר API של שיקוף ואיך ניגשים באמצעותו לכל אחד מהמודולים והסוגים של מכלל קוד, וקוראים את מאפייני זמן התכנון השונים של סוג נתונים. כמו כן נראה מספר שימושים מתקדמים בשיקוף, כמו הפעלה של שיטות ושימוש במידע סוג באופן דינמי (באמצעות כריכה מאוחרת) ואפילו יצירת וביצוע קוד MSIL בזמן ריצה!

## היררכיית API של השיקוף

היררכיית API של השיקוף מורכבת מסבכה של מחלקות המוגדרת במרחב השמות System.Reflection - תרשים 16-1 מציג חלק ממנה. מחלקות אלו מאפשרות גישה לוגית למכלל קוד ולמידע סוג. ניתן להתחיל מכל נקודה בהיררכיה זו כתלות בצרכים הייחודיים של כל יישום.




**תרשים 16-1** חלק מהיררכיית מחלקת השיקוף System.Reflection של NET

שים לב שמחלקות אלו מקיפות מידה גדולה של פונקציונליות. כאן לא אתמקד בפרטים של כל שיטה ושדה עבור כל מחלקה אלא אתן סקירה כללית על מחלקות המפתח ולאחר מכן אציג דוגמה הממחישה את הפונקציונליות שקרוב לוודאי תשתמש בה ביישומיך.

# המחלקה Type

בלב השיקוף נמצאת המחלקה המופשטת System.Type, המייצגת סוג במערכת סוגי הנתונים האוניברסלית (CTS) ומאפשרת לקבל מידע על שם הסוג, המודול ומרחב השמות בתוכם הוא נמצא והאם זהו סוג ייחוס (reference type) או סוג ערך (value type).

## קבלת אובייקט Type משם משתנה

הדוגמה הבאה מראה איך לקבל אובייקט Type עבור מופע (instance) של int:

```
using System;
using System.Reflection;

class TypeObjectFromInstanceApp
{
   public static void Main(string[] args)
   {
      int i = 6;
      Type t = i.GetType();
      Console.WriteLine(t.Name);
   }
}
```

## קבלת אובייקט Type משם של סוג

מלבד קבלת אובייקט Type של משתנה, ניתן גם ליצור אובייקט Type משם של סוג, כלומר, לא חייבים מופע של הסוג. כך לדוגמה עושים זאת עבור הסוג System.Int32:

```
using System;
using System.Reflection;

class TypeObjectFromNameApp
{
   public static void Main(string[] args)
   {
      Type t = Type.GetType("System.Int32");
      Console.WriteLine(t.Name);
   }
}
```

שים לב שלא ניתן להשתמש בכינויים כאשר קוראים לשיטה Type.GetType מאחר ושיטה זו משמשת את כל השפות. לכן לא ניתן להשתמש ב-int של C# במקום ב-System.Int32.

## Interrogating Types

המחלקה System.Type מאפשרת לאחזר מידע על כמעט כל אחד מאפיוני הסוג, כולל מציין הגישה שלו, האם הוא מקונן, תכונות COM שלו וכדומה. הקוד הבא ממחיש זאת באמצעות סוגים שכיחים וגם סוגי הדגמה:

```
using System;
using System.Reflection;

interface DemoInterface
{
}

class DemoAttr : System.Attribute
{
}

enum DemoEnum
{
}

public class DemoBaseClass
{
}
```

```
public class DemoDerivedClass : DemoBaseClass
{
}

class DemoStruct
{
}

class QueryTypesApp
{
    public static void QueryType(string typeName)
    {
        try
        {
            Type type = Type.GetType(typeName);
            Console.WriteLine("Type name: {0}", type.FullName);
            Console.WriteLine("\tHasElementType = {0}", type.HasElementType);
            Console.WriteLine("\tIsAbstract = {0}", type.IsAbstract);
            Console.WriteLine("\tIsAnsiClass = {0}", type.IsAnsiClass);
            Console.WriteLine("\tIsArray = {0}", type.IsArray);
            Console.WriteLine("\tIsAutoClass = {0}", type.IsAutoClass);
            Console.WriteLine("\tIsAutoLayout = {0}", type.IsAutoLayout);
            Console.WriteLine("\tIsByRef = {0}", type.IsByRef);
            Console.WriteLine("\tIsClass = {0}", type.IsClass);
            Console.WriteLine("\tIsCOMObject = {0}", type.IsCOMObject);
            Console.WriteLine("\tIsContextful = {0}", type.IsContextful);
            Console.WriteLine("\tIsEnum = {0}", type.IsEnum);
            Console.WriteLine("\tIsExplicitLayout = {0}", type.IsExplicitLayout);
            Console.WriteLine("\tIsImport = {0}", type.IsImport);
            Console.WriteLine("\tIsInterface = {0}", type.IsInterface);
            Console.WriteLine("\tIsLayoutSequential = {0}", type.IsLayoutSequential);
            Console.WriteLine("\tIsMarshalByRef = {0}", type.IsMarshalByRef);
            Console.WriteLine("\tIsNestedAssembly = {0}", type.IsNestedAssembly);
            Console.WriteLine("\tIsNestedFamANDAssem = {0}",
                              type.IsNestedFamANDAssem);
            Console.WriteLine("\tIsNestedFamily = {0}", type.IsNestedFamily);
```

```
        Console.WriteLine("\tIsNestedFamORAssem = {0}",
                    type.IsNestedFamORAssem);
        Console.WriteLine("\tIsNestedPrivate = {0}", type.IsNestedPrivate);
        Console.WriteLine("\tIsNestedPublic = {0}", type.IsNestedPublic);
        Console.WriteLine("\tIsNotPublic = {0}", type.IsNotPublic);
        Console.WriteLine("\tIsPointer = {0}", type.IsPointer);
        Console.WriteLine("\tIsPrimitive = {0}", type.IsPrimitive);
        Console.WriteLine("\tIsPublic = {0}", type.IsPublic);
        Console.WriteLine("\tIsSealed = {0}", type.IsSealed);
        Console.WriteLine("\tIsSerializable = {0}", type.IsSerializable);
        Console.WriteLine("\tIsServicedComponent = {0}",
                    type.IsServicedComponent);
        Console.WriteLine("\tIsSpecialName = {0}", type.IsSpecialName);
        Console.WriteLine("\tIsUnicodeClass = {0}", type.IsUnicodeClass);
        Console.WriteLine("\tIsValueType = {0}", type.IsValueType);
    }
    catch(System.NullReferenceException)
    {
        Console.WriteLine ("{0} is not a valid type", typeName);
    }
}

    public static void Main(string[] args)
    {
        QueryType("System.Int32");
        QueryType("System.Int64");
        QueryType("System.Type");

        QueryType("DemoAttr");
        QueryType("DemoEnum");
        QueryType("DemoBaseClass");
        QueryType("DemoDerivedClass");
        QueryType("DemoStruct");
    }
}
```

# Modules-ו Assemblies

מכללי קוד נידונים בהרחבה בפרק 18. למטרות הדיון כאן, מספיק לדעת שמכלל קוד (assembly) הוא קובץ פיסי המורכב מקבצי PE מרובים של NET. היתרון העיקרי של מכלל קוד הוא באפשרות לקבץ פונקציונליות על פי קירבה סמנטית לצרכי הטמעה ועדכוני גרסאות. המחלקה Assembly היא ייצוג זמן הריצה של מכלל קוד והיא גם נמצאת בראש היררכיית האובייקטים של השיקוף. ניתן לבצע משימות רבות בעזרת המחלקה Assembly. להלן כמה מהמשימות הנפוצות אותן נסקור בהמשך:

- ביצוע איטרציה על הסוגים של מכלל קוד.
- פירוט המודולים במכלל.
- קביעת מידע מזהה, כגון שם הקובץ הפיסי של המכלל ומיקומו.
- בדיקת מידע על גרסאות ואבטחה.
- אחזור נקודת הכניסה של המכלל.

## ביצוע איטרציה על הסוגים של מכלל קוד

בכדי לבצע איטרציה על הסוגים של מכלל קוד נתון, צריך רק ליצור מופע מאובייקט Assembly ואז לגשת למידע במערך Types המוחזר עבור מכלל זה. הנה דוגמה:

```
using System;
using System.Diagnostics;
using System.Reflection;

class DemoAttr : System.Attribute
{
}

enum DemoEnum
{
}

class DemoBaseClass
{
}
```

```
class DemoDerivedClass : DemoBaseClass
{
}

class DemoStruct
{
}

class GetTypesApp
{
    protected static string GetAssemblyName(string[] args)
    {
        string assemblyName;

        if (0 == args.Length)

        {
            Process p = Process.GetCurrentProcess();
            assemblyName = p.ProcessName + ".exe";
        }
        else
            assemblyName = args[0];

        return assemblyName;
    }

    public static void Main(string[] args)
    {
        string assemblyName = GetAssemblyName(args);

        Console.WriteLine("Loading info for " + assemblyName);
        Assembly a = Assembly.LoadFrom(assemblyName);

        Type[] types = a.GetTypes();
        foreach(Type t in types)
        {
            Console.WriteLine("\nType information for: " +
                              t.FullName);
            Console.WriteLine("\tBase class = " +
                              t.BaseType.FullName);
        }
    }
}
```

הערה

אם אתה מנסה לבצע קוד הדורש התרת אבטחה - כגון קוד המשתמש ב-API של שיקוף - דרך רשת אינטראנט, עליך לשנות מדיניות. דרך אחת לעשות זאת היא בעזרת הכלי Code Access Security Policy (caspol.exe). הנה דוגמה לשימוש בכלי זה:

```
caspol -addgroup 1.2 -url "file://somecomputer/someshare/*"
        SkipVerification
```

דוגמה זו מעניקה הרשאה נוספת - במקרה זה, SkipVerification - על סמך URL של הקוד המתבצע. שים לב שניתן לשנות מדיניות עבור כל הקוד הנמצא באזור נתון או רק עבור מכלל מסוים, על סמך חתימה מוצפנת או hash של המידע. ניתן לראות את הארגומנטים החוקיים של caspol.exe בהקלדת caspol -? בשורת הפקודה או באמצעות התיעוד המקוון של MSDN.

החלק הראשון של השיטה Main קובע אם העברת שם של מכלל קוד ליישום. אם לא, משתמשים בשיטה הסטטית getProcessName של המחלקה Process בכדי לקבוע את שם היישום המתבצע כרגע.

לאחר מכן מתחילים לראות עד כמה קלות רוב משימות השיקוף. הדרך הקלה ביותר ליצור מופע מאובייקט Assembly היא לקרוא לשיטה Assembly.LoadFrom. שיטה זו לוקחת ארגומנט יחיד: מחרוזת המייצגת את שמו של הקובץ הפיסי שאותו רוצים לטעון. הקריאה לשיטה Assembly.GetTypes מחזירה מערך אובייקטים מסוג Type. בנקודה זו יש בידינו אובייקט המתאר כל אחד ואחד מהסוגים הנמצאים במכלל והיישום מדפיס את מחלקת הבסיס שלו.

הנה הפלט המתקבל מביצוע יישום זה כאשר אתה מציין את הקובץ gettypes.exe או כאשר אינך מעביר כלל ארגומנטים ליישום:

```
Loading info for GetTypes.exe

Type information for: DemoAttr
   Base class = System.Attribute

Type information for: DemoEnum
   Base class = System.Enum

Type information for: DemoBaseClass
   Base class = System.Object

Type information for: DemoDerivedClass
   Base class = DemoBaseClass
```

```
Type information for: DemoStruct
   Base class = System.Object

Type information for: AssemblyGetTypesApp
   Base class = System.Object
```

## פירוט המודולים של מכלל קוד

למרות שמרבית היישומים בספר זה מורכבים ממודול אחד, ניתן ליצור מכללי קוד המורכבים ממודולים מרובים. ניתן לאחזר מידע על מודולים מאובייקט Assembly בשתי דרכים. הדרך הראשונה היא לבקש מערך של כל המודולים. כך ניתן לגשת לכל אחד מהם באמצעות לולאה ולקבל כל מידע נדרש. הדרך השנייה היא לקבל מודול מסוים. הבה נבחן כל אחת מגישות אלו.

בכדי לבצע איטרציה על מודולים של מכלל, יש ליצור מכלל עם יותר ממודול אחד. נעשה זאת על ידי העברת GetAssemblyName לתוך מחלקה משלה והכנסת מחלקה זו לתוך קובץ בשם AssemblyUtils.netmodule:

```
using System.Diagnostics;

namespace MyUtilities
{
   public class AssemblyUtils
   {
      public static string GetAssemblyName(string[] args)
      {
         string assemblyName;

         if (0 == args.Length)
         {
            Process p = Process.GetCurrentProcess();
            assemblyName = p.ProcessName + ".exe";
         }
         else
            assemblyName = args[0];

         return assemblyName;
      }
   }
}
```

לאחר מכן נבנה את netmodule באמצעות הפקודה הבאה:

```
csc /target:module AssemblyUtils.cs
```

המתג target:module/ גורם למהדר ליצור מודול שיוכלל מאוחר יותר במכלל קוד. הפקודה שלעיל תיצור קובץ בשם AssemblyUtils.netmodule. בפרק 18 אסקור ביתר פירוט את האפשרויות השונות כאשר יוצרים מכללים ומודולים.

בנקודה זו, אנו רוצים ליצור מודול שני כך שיהיה לנו משהו לשקף ממנו. להלן היישום המשתמש במחלקה AssemblyUtils. שים לב למשפט using המתייחס למרחב השמות MyUtilities.

```
using System;
using System.Reflection;
using MyUtilities;

class GetModulesApp
{
   public static void Main(string[] args)
   {
      string assemblyName = AssemblyUtils.GetAssemblyName(args);

      Console.WriteLine("Loading info for " + assemblyName);
      Assembly a = Assembly.LoadFrom(assemblyName);

      Module[] modules = a.GetModules();
      foreach(Module m in modules)
      {
         Console.WriteLine("Module: " + m.Name);
      }
   }
}
```

בכדי לצרף את AssemblyUtils.netmodule למכלל הקוד של היישום עלינו להשתמש במתגי המהדר הבאים עבור שורת הפקודה:

```
csc /addmodule:AssemblyUtils.netmodule GetModules.cs
```

כעת קיבלנו מכלל קוד עם שני מודולים שונים. בכדי לראות זאת נבצע את היישום ונקבל את הפלט הבא:

```
Loading info for GetModulesApp.exe
Module: GetModulesApp.exe
Module: AssemblyUtils.netmodule
```

כפי שרואים, קוד זה פשוט יוצר מופע מאובייקט Assembly וקורא לשיטה GetModules שלו. מנקודה זו מתבצעת איטרציה על פני המערך המוחזר ומודפס שמו של כל אחד מהמודולים.

# Late Binding with Reflection

לפני מספר שנים עבדתי עבור חטיבת המולטימדיה של חברת IBM על מוצר בשם IBM/World Book Multimedia Encyclopedia. אחד האתגרים שלנו היה לכתוב יישום שיאפשר למשתמש להגדיר פרוטוקולים שונים של תקשורת לשרתים של World Book. הפתרון היה צריך להיות דינמי כי למשתמש היתה אפשרות להוסיף ולהסיר ללא הרף פרוטוקולים שונים (לדוגמה, TCP/IP, IGN, CompuServ וכדומה) אל המערכת או ממנה. יחד עם זאת, היישום נדרש "לדעת" אלה פרוטוקולים היו קיימים כך שהמשתמש יוכל לבחור בפרוטוקול מסוים להגדרה ולשימוש.

הפתרון שהגענו אליו היה ליצור קבצי DLL עם סיומת מיוחדת ולהתקין אותם במחיצת היישום. כאשר המשתמש רצה לראות רשימה של פרוטוקולים מותקנים, היישום היה קורא לפונקציה LoadLibrary של Win32 בכדי לטעון כל DLL ואז קורא לפונקציה GetProcessAddress בכדי לקבל מצביע פונקציה לפונקציה הרצויה. זו דוגמה מצוינת של **כריכה מאוחרת** (late binding) בתכנות סטנדרטי ל- Win32 בכך שהמהדר אינו יודע דבר אודות קריאות אלו בזמן הבנייה. כפי שתראה בדוגמה הבאה, ב-NET ניתן לבצע את המשימה באמצעות המחלקה Assembly, שיקוף סוגים, ומחלקה חדשה הנקראת Activator.

ניצור מחלקה מופשטת בשם CommProtocol ונגדיר אותה ב-DLL משלה כך שניתן יהיה לשתפה בין קבצי DLL מרובים שרוצים לגזור ממנה. (הפרמטרים של שורת הפקודה משובצים בהערות הקוד).

```
// CommProtocol.cs
// Build with the following command line switches
//        csc /t:library commprotocol.cs
public abstract class CommProtocol
{
   public static string DLLMask = "CommProtocol*.dll";
   public abstract void DisplayName();
}
```

כעת ניצור שני קבצי DLL נפרדים. כל DLL מייצג פרוטוקול תקשורות ומכיל מחלקה הנגזרת מהמחלקה המופשטת CommProtocol. שים לב ששניהם צריכים להתייחס ל-CommProtocol.dll בזמן ההידור. הנה DLL עבור הפרוטוקול IGN:

```
// CommProtocolIGN.cs
// Build with the following command line switches
//        csc /t:library CommProtocolIGN.cs /r:CommProtocol.dll
using System;
```

```
public class CommProtocolIGN : CommProtocol
{
   public override void DisplayName()
   {
      Console.WriteLine("This is the IBM Global Network");
   }
}
```

והנה DLL עבור הפרוטוקול TCP/IP:

```
// CommProtocolTcpIp.cs
// Build with the following command line switches
//        csc /t:library CommProtocolTcpIp.cs /r:CommProtocol.dll
using System;

public class CommProtocolTcpIp : CommProtocol
{
   public override void DisplayName()
   {
      Console.WriteLine("This is the TCP/IP protocol");
   }
}
```

כעת נראה עד כמה קל לטעון באופן דינמי מכלל קוד, לחפש סוג, ליצור מופע ממנו ולקרוא לאחת משיטותיו.

```
using System;
using System.Reflection;
using System.IO;

class LateBindingApp
{
   public static void Main()
   {

      string[] fileNames = Directory.GetFiles
                           (Environment.CurrentDirectory,
                            CommProtocol.DLLMask);
      foreach(string fileName in fileNames)
      {
         Console.WriteLine("Loading DLL '{0}'", fileName);
```

```
        Assembly a = Assembly.LoadFrom(fileName);

        Type[] types = a.GetTypes();
        foreach(Type t in types)
        {
            if (t.IsSubclassOf(typeof(CommProtocol)))
            {
                object o = Activator.CreateInstance(t);

                MethodInfo mi = t.GetMethod("DisplayName");

                Console.Write("\t");
                mi.Invoke(o, null);
            }
            else
            {
                Console.WriteLine("\tThis DLL does not have " +
                "CommProtocol-derived class defined");
            }
        }
    }
  }
}
```

תחילה השתמשנו במחלקה System.IO.Directory בכדי למצוא את כל קבצי DLL עם המסכה CommProtocol*.dll במחיצה הנוכחית. השיטה Directory.GetFiles מחזירה מערך של אובייקטים מסוג string המייצג את שמות הקבצים התואמים לקריטריון החיפוש. לאחר מכן ניתן להשתמש בלולאת foreach בכדי לבצע איטרציה על פני המערך וקריאה לשיטה Assembly.LoadFrom עליה למדת קודם בפרק זה. ברגע שנוצר מכלל עבור DLL נתון, ביצענו איטרציה על פני כל הסוגים של המכלל וקראנו לשיטה Type.SubClassOf בכדי לקבוע האם המכלל מכיל סוג הנגזר מ-CommProtocol. אנו מניחים שאם מוצאים אחד מאלה אזי יש לנו DLL חוקי שאפשר לעבוד איתו. כאשר אכן מוצאים DLL שמכיל סוג הנגזר מ-CommProtocol, יוצרים מופע מאובייקט Activator ומעבירים לבנאי שלו את האובייקט מסוג Type. כפי ששמה מרמז, המחלקה Activator משמשת ליצירה (או הפעלה) דינמית של סוג.

לאחר מכן השתמשנו בשיטה Type.GetMethod ליצירת אובייקט MethodInfo, עם ציון שם השיטה DisplayName. מרגע שעשינו זאת אפשר להשתמש בשיטה Invoke של האובייקט MethodInfo, להעביר לה את הסוג המופעל והרי השיטה DisplayName של ה-DLL.

# יצירה וביצוע קוד בזמן ריצה

לאחר שראינו איך משקפים סוגים בזמן ריצה, מגדירים כריכה מאוחרת לקוד ומבצעים קוד באופן דינמי, נמשיך עם אותו ההיגיון וניצור קוד בזמן ריצה. יצירת קוד בזמן ריצה כרוכה בשימוש במרחב השמות System.Reflection.Emit. השימוש במחלקות במרחב שמות זה מאפשר להגדיר מכלל קוד בזיכרון, ליצור מודול עבור מכלל קוד, להגדיר סוגים חדשים עבור מודול (כולל החברים שלו), ואפילו לפלוט (emit) את קודי הפעולה של MSIL עבור לוגיקת היישום.

למרות שהקוד בדוגמה הבאה הוא מאד פשוט, הפרדתי את קוד השרת - DLL המכיל מחלקה יוצרת-קוד (היוצרת שיטה בשם HelloWorld) - מקוד הלקוח, יישום היוצר מופע מהמחלקה יוצרת-הקוד וקורא לשיטה HelloWorld שלה. (שים לב שמתגי המהדר נמצאים בהערות הקוד). הנה קוד DLL:

```
using System;
using System.Reflection;
using System.Reflection.Emit;

namespace ILGenServer
{
   public class CodeGenerator
   {
      public CodeGenerator()
      {
         // Get current currentDomain.
         currentDomain = AppDomain.CurrentDomain;

         // Create assembly in current currentDomain.
         assemblyName = new AssemblyName();
         assemblyName.Name = "TempAssembly";

         assemblyBuilder =
               currentDomain.DefineDynamicAssembly
               (assemblyName, AssemblyBuilderAccess.Run);

         // create a module in the assembly
         moduleBuilder = assemblyBuilder.DefineDynamicModule
                        ("TempModule");

         // create a type in the module
         typeBuilder = moduleBuilder.DefineType
                      ("TempClass",
                      TypeAttributes.Public);
         // add a member (a method) to the type
         methodBuilder = typeBuilder.DefineMethod
                        ("HelloWorld",
                         MethodAttributes.Public,
                         null,null);
```

```
            // Generate MSIL.
            msil = methodBuilder.GetILGenerator();
            msil.EmitWriteLine("Hello World");
            msil.Emit(OpCodes.Ret);

            // Last "build" step : create type.
            t = typeBuilder.CreateType();
        }

        AppDomain currentDomain;
        AssemblyName assemblyName;
        AssemblyBuilder assemblyBuilder;
        ModuleBuilder moduleBuilder;
        TypeBuilder typeBuilder;
        MethodBuilder methodBuilder;
        ILGenerator msil;
        object o;

        Type t;
        public Type T
        {
            get
            {
                return this.t;
            }
        }
    }
}
```

תחילה יצרנו מופע מאובייקט AppDomain מהתחום (domain) הנוכחי. (בפרק 17 תראה שתחומי היישום זהים מבחינה פונקציונלית לתהליכים של Win32). לאחר מכן יצרנו מופע מאובייקט AssemblyName. המחלקה AssemblyName נסקרת בהרחבה בפרק 18. כאן אסביר בקצרה שמחלקה זו מנוצלת על ידי מנהל מטמון המכלל לשליפת מידע אודות מכלל. מרגע שיש לנו את תחום היישום הנוכחי ושם מכלל מאותחל, אנו יכולים לקרוא לשיטה AppDomain.DefineDynamicAssembly ליצירת מכלל קוד חדש. אנו מעבירים לשיטה זו שני ארגומנטים: שם המכלל ומצב הגישה אליו. AssemblyBuilderAccess.Run מציין שהמכלל ניתן לביצוע בתוך הזיכרון אך לא ניתן לשמור אותו. השיטה AppDomain.DefineDynamicAssembly מחזירה אובייקט AssemblyBuilder המשתמע לאחר מכן כאובייקט Assembly. בנקודה זו יש לנו בזיכרון מכלל קוד פועל בשלמותו. כעת עלינו ליצור את המודול הזמני שלו ואת סוג מודול זה.

נתחיל עם קריאה לשיטה Assembly.DefineDynamicModule לקבלת אובייקט ModuleBuilder. ברגע שיש בידינו אובייקט זה, אנו קוראים לשיטה DefineType שלו ליצירת אובייקט TypeBuilder, כשאנו מעבירים לה את שם הסוג ("TempClass") ואת התכונות המשמשות להגדרתו (TypeAttributes.Public). עם אובייקט TypeBuilder זה נוכל ליצור כל סוג חבר שנחפוץ. במקרה זה אנו יוצרים שיטה באמצעות השיטה TypeBuilder.DefineMethod.

קיבלנו סוג חדש לגמרי בשם TempClass שבתוכו משובצת שיטה בשם HelloWorld. כעת עלינו להחליט איזה קוד יש להציב בתוך שיטה זו. לשם כך הקוד יוצר מופע מהאובייקט ILGenerator באמצעות השיטה MethodBuilder.GetILGenerator וקורא לשיטות ILGenerator השונות לכתיבת קוד MSIL לתוך השיטה.

שים לב שניתן להשתמש בקוד סטנדרטי כמו Console.WriteLine באמצעות שיטות ILGenerator השונות, או שניתן לפלוט קודי פעולה של MSIL באמצעות השיטה ILGenerator.Emit. שיטה זו לוקחת כארגומנט יחיד שדה חבר של המחלקה OpCodes המתייחס ישירות לקוד פעולה MSIL.

לסיום, אנו קוראים לשיטה TypeBuilder.CreateType. זה צריך להיות תמיד השלב האחרון שיש לבצע לאחר שמגדירים את החברים עבור סוג חדש. לאחר מכן אנו מקבלים אובייקט Type עבור הסוג החדש באמצעות השיטה Type.GetType. אובייקט זה מאוחסן במשתנה חבר לקריאת מידע מאוחר יותר על ידי יישום הלקוח.

כעת, כל שהלקוח נדרש לעשות הוא לקבל את החבר Type של CodeGenerator (סוג Type שנוצר באופן דינמי), ליצור מופע של Activator ולהעביר לבנאי שלו את אובייקט הסוג (להפעיל את הסוג), ליצור מופע אובייקט MethodInfo מהסוג, ואז לקרוא לשיטה (עם הסוג המופעל). הנה הקוד שעושה זאת, בתוספת קטנה של קוד לבדיקת שגיאות בכדי להבטיח שהכל יפעל כראוי:

```
using System;
using System.Reflection;
using ILGenServer;

public class ILGenClientApp
{
   public static void Main()
   {
      Console.WriteLine("Calling DLL function to generate " +
                        "a new type and method in memory...");
      CodeGenerator gen = new CodeGenerator();

      Console.WriteLine("Retrieving dynamically generated type...");
      Type t = gen.T;
      if (null != t)
      {
         Console.WriteLine("Instantiating the new type...");
         object o = Activator.CreateInstance(t);
```

```
            Console.WriteLine("Retrieving the type's " +
                              "HelloWorld method...");
            MethodInfo helloWorld = t.GetMethod("HelloWorld");
            if (null != helloWorld)
            {
                Console.WriteLine("Invoking our dynamically " +
                                  "created HelloWorld method...");
                                  helloWorld.Invoke(o, null);
            }
            else
            {
                Console.WriteLine("Could not locate " +
                                  "HelloWorld method");
            }
        }
        else
        {
            Console.WriteLine("Could not access Type from server");
        }
    }
}
```

אם תבנה ותבצע יישום זה תקבל את הפלט הבא:

```
Calling DLL function to generate a new type and method in memory...
Retrieving dynamically generated type...
Instantiating the new type...
Retrieving the type's HelloWorld method...
Invoking our dynamically created HelloWorld method...
Hello World
```

# סיכום

שיקוף הוא היכולת לגלות מידע על סוג בזמן ריצה. API של שיקוף מאפשר לך לבצע משימות כמו איטרציה על מודולים ועל סוגים של מכלל קוד, ולקרוא את מאפייני זמן התכנון השונים של סוג. שימושים מתקדמים בשיקוף כוללים הפעלה של שיטות ושימוש בסוגים באופן דינמי (באמצעות כריכה מאוחרת), ואפילו יצירה וביצוע קוד MSIL בזמן ריצה.

# פרק 17 Unmanaged Code

תוחלת החיים של שפה או סביבת פיתוח חדשה תהיה קצרה אם היא תתעלם ממערכות קיימות ותספק אמצעים לכתיבת מערכות חדשות בלבד. מתכנני הטכנולוגיה החדשה, מצוינת כל שתהיה, חייבים להתחשב בכך שהקוד החדש חייב לפעול זמן מה לצד הקוד הקיים. לשם כך הקלו מתכנני NET ו-C# על מתכנתים להפעיל קוד קיים בשיתוף עם קוד NET באמצעות **קוד לא מנוהל** (unmanaged code), המתייחס לקוד שאינו מנוהל, או אינו מבוקר על ידי זמן הריצה של NET. בפרק זה אסקור את שלוש הדוגמאות הבאות של קוד לא מנוהל:

- **שירותי הפעלת פלטפורמות** (Platform Invocation Services) - שירותים אלה מאפשרים לקוד NET לגשת לפונקציות, מבנים ואפילו פונקציות משוב (callbacks) בספריות קישור דינאמי לא מנוהלות קיימות (DLLs).
- **קוד לא בטוח** (Unsafe code) - כתיבת קוד לא בטוח מאפשרת למתכנת #C להשתמש במצביעים ביישומי #C במחיר של ניהול הקוד על ידי זמן הריצה של NET.
- **תפעוליות שיתופית עם רכיבי COM** (COM interoperability) - מונח זה מתייחס ליכולת של קוד NET להשתמש ברכיבי COM ושל יישומי COM להשתמש ברכיבי NET.

# שירותי הפעלת פלטפורמות

שירותי הפעלת פלטפורמות של NET - מוכרים גם כתוכנית PInvoke - מאפשרים לקוד מנוהל לעבוד עם פונקציות ומבנים שיוצאו מ-DLL. בחלק זה נלמד איך לקרוא לפונקציות DLL ונסקור את התכונות המשמשות להסדרת נתונים בין יישום NET לבין DLL.

מאחר ומהדר #C אינו מקבל את קוד המקור של פונקציית DLL, חייבים להגדיר לו את החתימה של שיטת המקור וכן מידע אודות כל ערך מוחזר, ואיך להסדיר (marshall) את הפרמטרים ל-DLL.

כידוע לך, ניתן ליצור קבצי DLL באמצעות מהדר #C ומהדרי NET אחרים. בחלק זה נמנעתי משימוש במונח "Win32 DLL לא מנוהל", לכן בכל פעם שאני מתייחס ל-DLL עליך להניח שהכוונה למרכיב לא מנוהל.

## הצהרה על פונקציית DLL חיצונית

הנושא הראשון אותו נלמד הוא איך מצהירים על פונקציית DLL פשוטה ב-#C. נתחיל במה שהופך במהירות להיות "HelloWorld" של תוכנית PInvoke, דוגמת "MessageBox" של Win32. לאחר מכן נעבור לתחומים מתקדמים יותר של הסדרת פרמטרים.

כפי שלמדת בפרק 8, תכונות (attributes) הן אמצעי לספק מידע זמן-תכנון עבור סוג של #C. מידע זה ניתן לאחזור בזמן-ריצה באמצעות שיקוף (reflection). #C משתמשת בתכונה הנקראת DllImport בכדי להעביר למהדר את תיאור פונקציית DLL שהיישום עתיד לקרוא לה. הנה התחביר של תכונה זו:

```
[DllImport(dllName)]
accessModifier static extern retValue dllFunction ( param1, param2,...) ;
```

כפי שרואים, ייבוא של פונקציית DLL נעשה בקלות על ידי הצמדת התכונה DllImport (מעבירים את שם DLL לבנאי שלה) לפונקציה החיצונית שלה רוצים לקרוא. שים לב במיוחד לעובדה שחייבים להשתמש גם במציינים static ו-extern בהגדרת הפונקציה. הנה הדוגמה MessageBox הממחישה עד כמה קל להשתמש ב-PInvoke:

```
using System;
using System.Runtime.InteropServices;
```

```
class PInvoke1App
{
   [DllImport("user32.dll")]
   static extern int MessageBoxA(int hWnd,
                                 string msg,
                                 string caption,
                                 int type);

   public static void Main()
   {
      MessageBoxA(0,
                  "Hello, World!",
                  "This is called from a C# app!",
                  0);
   }
}
```

ביצוע היישום מקפיץ את תיבת ההודעה הצפויה עם המשפט "Hello, World!".

שים לב למשפט using עם ייחוס למרחב השמות System.Runtime.InteropServices. מרחב שמות זה הוא אשר מגדיר את התכונה DllImport. לאחר מכן הגדרתי שיטה בשם MessageBoxA בתוך השיטה Main. עם זאת, כאשר רוצים לקרוא לשיטת C# הפנימית בשם שונה מזה של הפונקציה DLL משתמשים באחד מהפרמטרים לפי שם של התכונה DllImport.

הקוד הבא אומר למהדר להשתמש בשם MessageBoxA עבור השיטה. מאחר ולא הגדרתי במפורש את שמה של הפונקציה DLL בתכונה DllImport, המהדר מניח ששני השמות זהים.

```
[DllImport("user32.dll")]
   static extern int MessageBoxA(int hWnd,
                                 string msg,
                                 string caption,
                                 int type);
```

כעת נשנה את התנהגות ברירת המחדל ונשתמש בשם פנימי MsgBox בכדי לקרוא לאותה פונקציית DLL.

```
using System;
using System.Runtime.InteropServices;
```

```
class PInvoke2App
{
   [DllImport("user32.dll", EntryPoint="MessageBoxA")]
   static extern int MsgBox(int hWnd,
                            string msg,
                            string caption,
                            int type);

   public static void Main()
   {
      MsgBox(0,
            "Hello, World!",
            "This is called from a C# app!",
            0);
   }
}
```

כפי שרואים, יש לציין את הפרמטר לפי שם EntryPoint של התכונה DllImport בכדי שנוכל לכנות את השיטה השקולה לפונקציית DLL חיצונית בכל שם שנרצה.

לסיום נסקור את הפרמטר CharSet. פרמטר זה מאפשר להגדיר את קבוצת התווים שבה משתמש הקובץ DLL. כאשר כותבים יישומי C++, בדרך כלל לא מציינים במפורש את גרסת קבוצת התווים של הפונקציה (MessageBoxA או MessageBoxW) מאחר והמהדר כבר יודע באיזו קבוצת תווים - ANSI או Unicode - אתה משתמש. לכן ב- C++ מספיק לקרוא לפונקציה MessageBox והמהדר יקבע לאיזו גירסה של קבוצת תווים לקרוא. באופן דומה, ב-C# מגדירים את קבוצת היעד של התווים לתכונה DllImport וקבוצה זו תציין לאיזו גירסה של MessageBox יש לקרוא. בדוגמה הבאה העברתי ל-DllImport את הערך CharSet.Ansi דרך הפרמטר לפי שם CharSet של התכונה ולכן עדיין תהיה קריאה לפונקציה MessageBoxA.

```
using System;
using System.Runtime.InteropServices;

class PInvoke3App
{
   // CharSet.Ansi will result in a call to MessageBoxA.
   // CharSet.Unicode will result in a call to MessageBoxW.
   [DllImport("user32.dll", CharSet=CharSet.Ansi)]
   static extern int MessageBox(int hWnd,
                            string msg,
                            string caption,
                            int type);
```

```
    public static void Main()
    {
        MessageBox(0,
                "Hello, World!",
                "This is called from a C# app!",
                0);
    }
}
```

יתרון השימוש בפרמטר CharSet הוא בכך שניתן להגדיר משתנה ביישום הקובע איזו גירסה של הפונקציות (ANSI או Unicode) נקראת. אין צורך לשנות את כל הקוד אם רוצים להחליף בין הגרסאות.

## PInvoke-ו Callbacks

פונקציית DLL יכולה גם כן לקרוא לשיטות מוגדרות ביישום #C בתסריטים של קריאות חוזרות (callbacks). תסריטים אלה מורכבים משימוש בכל אחת מפונקציות EnumXXX של Win32 ובהם קוראים לפונקציה שתפקידה למנות אלמנטים כלשהם. לפונקציה כזו מעבירים מצביע פונקציה אשר ייקרא על ידי Windows עבור כל פריט שנמצא. הדבר נעשה באמצעות צירוף של PInvoke - בכדי לקרוא לפונקציית DLL - ונציגים - להגדרת הקריאה החוזרת. תוכל לעיין שוב בפרק 14 בכדי לרענן את ידיעותיך בנושא נציגים.

הקוד הבא מונה ומדפיס את הכותרות של כל החלונות במערכת:

```
using System;
using System.Runtime.InteropServices;
using System.Text;

class CallbackApp
{
 [DllImport("user32.dll")]
 static extern int GetWindowText(int hWnd, StringBuilder text, int count);

 delegate bool CallbackDef(int hWnd, int lParam);

 [DllImport("user32.dll")]
 static extern int EnumWindows (CallbackDef callback, int lParam);
```

```
 static bool PrintWindow(int hWnd, int lParam)
   {
      StringBuilder text = new StringBuilder(255);
      GetWindowText(hWnd, text, 255);

      Console.WriteLine("Window caption: {0}", text);
      return true;
   }

   static void Main()
   {
      CallbackDef callback = new CallbackDef(PrintWindow);
      EnumWindows(callback, 0);
   }
}
```

תחילה מגדירים את הפונקציות EnumWindows ו-GetWindowText של Win32 באמצעות התכונה DllImport. לאחר מכן נגדיר נציג בשם CallbackDef ושיטה בשם PrintWindows. כעת כל שנדרש הוא ליצור מופע מהנציג CallbackDef בשיטה Main (מעבירים לו את השיטה PrintWindows) ולקרוא לשיטה EnumWindows. Windows תקרא לשיטה PrintWindows עבור כל חלון שקיים במערכת.

השיטה PrintWindows מעניינת בכך שהיא משתמשת במחלקה StringBuilder ליצירת מחרוזת בגודל קבוע המועברת לפונקציה GetWindowText. זו הסיבה לכך שהפונקציה GetWindowText מוגדרת כך:

```
static extern int GetWindowText(int hWnd, StringBuilder text, int count);
```

בכל אופן, הסיבה לכך היא שפונקציית DLL אינה רשאית לשנות string, ולכן אינך יכול להשתמש בסוג זה. גם אם תנסה להעביר לה string לפי ייחוס, אין לקוד הקורא כל דרך לאתחל string בגודל הנכון.

## Marshalling ו- PInvoke

למרות שבדרך כלל אינך רואה את ההסדרה (marshalling) או מגדיר איך היא תפעל, בכל פעם שקוראים לפונקציית DLL, NET צריכה להסדיר (או לארגן) את הפרמטרים לפונקציה ואת הערך המוחזר ליישום NET הקורא לה. בדוגמאות הקודמות לא נדרשנו לעשות דבר לשם כך היות ו-NET מגדירה ערך מקור של ברירת מחדל עבור כל סוג של NET. לדוגמה, הפרמטרים השני והשלישי לפונקציות MessageBoxA ו-MessageBoxW הוגדרו מסוג string. עם זאת, מהדר #C יודע ש-string של #C הוא שווה-ערך ל-LPSTR של Win32. ואולם מה קורה אם רוצים לדרוס את התנהגות ההסדרה של ברירת המחדל? לשם כך משתמשים בתכונה MarshalAs המוגדרת גם כן במרחב השמות System.Runtime.InteropServices.

בדוגמה הבאה אני שוב משתמש ב-MessageBox לשם הפשטות. כזכור לך מסעיף קודם, צריך רק להגדיר את הקבוע CharSet.Unicode עבור הפרמטר לפי שם CharSet של התכונה DllImport. עם זאת, כאן אני רוצה שהמהדר יסדיר את הנתונים כ-LPWSTR, ולכן אני משתמש בתכונה MarshalAs ומגדיר עם הקבוע הממוספר UnmanagedType לאיזה סוג אני רוצה להמיר את הסוג שלי.

```
using System;
using System.Runtime.InteropServices;

class PInvoke4App
{
   [DllImport("user32.dll", CharSet=CharSet.Unicode)]
   static extern int MessageBox(int hWnd,
                        [MarshalAs(UnmanagedType.LPWStr)]
                        string msg,
                        [MarshalAs(UnmanagedType.LPWStr)]
                        string caption,
                        int type);

   public static void Main()
   {
      MessageBox(0,
               "Hello, World!",
               "This is called from a C# app!",
               0);
   }
}
```

שים לב שניתן להצמיד את התכונה MashalAs לפרמטרים של שיטה, לערכים מוחזרים של שיטה ולשדות של מבנים ומחלקות. כמו כן, שים לב שבכדי לשנות את הסדרת ברירת המחדל עבור ערך מוחזר של שיטה, צריך להצמיד את התכונה MarshalAs לשיטה עצמה.

# Unsafe Code

בקרב חלק מהמתכנתים העוברים מ- C++ ל-C# שוררת דאגה לגבי היכולת לקבל "שליטה מלאה" על טיפול בזיכרון במקרים שבהם זקוקים לכך, סוגיה המתייחסת לקוד לא בטוח. למרות השם המפחיד, **קוד לא בטוח** (unsafe code) אינו קוד לא בטוח או לא אמין מטבעו - זהו קוד עבורו זמן הריצה של NET אינו מפקח על הקצאה ופינוי של זיכרון. היכולת לכתוב קוד לא בטוח מועילה ביותר כאשר משתמשים במצביעים בכדי לתקשר עם קוד קיים (כגון API של C) או כאשר היישום מחייב טיפול ישיר בזיכרון (בדרך כלל משיקולי ביצועים).

כותבים קוד לא בטוח באמצעות שתי מילות מפתח: unsafe ו-fixed. המילה unsafe מציינת שהבלוק המסומן יבוצע בהקשר לא מנוהל. ניתן להחיל מילת מפתח זו על כל השיטות, כולל בנאים ומאפיינים, ואפילו על בלוקים של קוד בתוך שיטות. המילה fixed אחראית ל**קיבוע** (pinning) של אובייקטים לא מנוהלים. קיבוע הוא הפעולה המונעת מ-garbage collector להזיז את האובייקט מכתובתו בזיכרון. מה שקורה הוא שבמהלך ביצועו של יישום, מתבצעים הקצאה ופינוי של אובייקטים ונפערים "חורים" בזיכרון. במקום לגרום לזיכרון מקוטע, זמן הריצה של NET מעביר את האובייקטים ממקום למקום בכדי להביא לניצול מירבי של זיכרון. כמובן שאם יש לך מצביע לכתובת זיכרון מסוימת, אין זה רצוי שזמן הריצה יעביר את האובייקט מכתובת זו ללא ידיעתך, ובכך ישאיר אותך עם מצביע חסר תוקף. היות ו-garbage collector מעביר אובייקט בזיכרון בכדי להגדיל את יעילות היישום, עליך להשתמש במילת מפתח זו בתבונה.

## Pointers

הבה נסתכל על מספר חוקים ביחס לשימוש במצביעים וקוד לא בטוח ב-#C, ולאחר מכן נעבור למספר דוגמאות. ניתן לרכוש מצביעים רק עבור סוגי ערך, מערכים ומחרוזות. במקרה של מערכים, האלמנט הראשון חייב להיות מסוג ערך מאחר ו-#C מחזירה למעשה מצביע לאלמנט הראשון של המערך ולא למערך עצמו. לפיכך, מנקודת המבט של המהדר, #C מחזירה מצביע לסוג ערך ולא לסוג ייחוס.

טבלה 17-1 מראה איך הסמנטיקה הסטנדרטית של מצביעי C/C++ נתמכת ב-#C.

**טבלה 17-1** האופרטורים של מצביעי C/C++

| אופרטור | תיאור |
|---|---|
| & | אופרטור **כתובת-של** (address-of) מחזיר מצביע המייצג את הכתובת בזיכרון של משתנה. |
| * | אופרטור **עקיפה** (dereference) (מוכר גם כאופרטור חילוץ כתובת) מחזיר מצביע לכתובת של הערך. |
| -> | אופרטור **עקיפה וגישה לחבר** (dereferencing and member access) (מוכר גם כמצביע-אל) מחזיר מצביע ומאפשר גישה לא ישירה לחבר . |

הדוגמה הבאה נראית מוכרת למפתחים ב-C או ++C. כאן קוראים לשיטה המקבלת שני מצביעים למשתנים מסוג int ומשנה את ערכיהם לפני שהיא חוזרת לשיטה הקוראת. דוגמה זו, על אף שאינה מלהיבה במיוחד, אכן ממחישה את השימוש במצביעים ב-#C:

```
// Compile this application with the /unsafe option.

using System;

class Unsafe1App
{
   public static unsafe void GetValues(int* x, int* y)
   {
      *x = 6;
      *y = 42;
   }

   public static unsafe void Main()
   {
      int a = 1;
      int b = 2;
      Console.WriteLine("Before GetValues() : a = {0}, b = {1}",
                        a, b);
      GetValues(&a, &b);
      Console.WriteLine("After GetValues() : a = {0}, b = {1}",
                        a, b);
   }
}
```

יש להדר דוגמה זו עם האופציה unsafe/ של המהדר. הנה הפלט המתקבל מיישום זה:

```
Before GetValues() : a = 1, b = 2
After GetValues() : a = 6, b = 42
```

## המשפט fixed

המשפט fixed הוא בעל התחביר הבא:

**fixed** (type * ptr = expression) statement

כפי שציינתי, המשפט אומר ל-garbage collector להתעלם ממשתנה מסוים. בתחביר זה המילה type מתייחסת לסוג לא מנוהל או void, expression הוא כל ביטוי שתוצאתו היא מצביע type, ו-statement מתייחס לבלוק הקוד עליו חל קיבוע הקוד.

להלן דוגמה פשוטה:

```
using System;

class Foo
{
   public int x;
}

class Fixed1App
{
   unsafe static void SetFooValue(int* x)
   {
      Console.WriteLine("Dereferenced pointer to modify foo.x");
      *x = 42;
   }

   unsafe static void Main()
   {
      // Create an instance of the structure.
      Console.WriteLine("Creating the Foo class");
      Foo foo = new Foo();

      Console.WriteLine("foo.x intialized to {0}", foo.x);

      // The fixed statement pins the foo object until
      // the enclosing compound statement ends.
      Console.WriteLine("Setting pointer to foo.x");
      // Assign the address of the foo object to a Foo*.
      fixed(int* f = &foo.x)
      {
         Console.WriteLine("Calling SetFooValue passing " +
                           "pointer to foo.x");
         SetFooValue(f);
      }

      // Show that we did alter the member via its pointer.
      Console.WriteLine("After return from " +
                        "SetFooValue, foo.x = {0}", foo.x);
   }
}
```

קוד זה יוצר מופע מאובייקט מחלקה בשם Foo ומקבע אובייקט זה בתוך משפט fixed בזמן שהוא מציב את כתובת החבר הראשון שלו למשתנה מסוג int* (הסוג הנדרש עבור השיטה SetFooValue). שים לב שהמשפט Fixed משמש להקפת אותו קטע קוד

שיושפע במידה ו-garbage collector יעביר את האובייקט Foo. זו סוגיה עדינה אך חשובה עבור בלוקים גדולים וארוכים יותר של קוד, שם רוצים למזער את משך הזמן שבו האובייקט מקובע. הנה הפלט המתקבל לאחר הידור וביצוע היישום:

```
Creating the Foo class
foo.x intialized to 0
Setting pointer to foo.x
Calling SetFooValue passing pointer to foo.x
Dereferenced pointer to modify foo.x
After return from SetFooValue, foo.x = 42
```

הערה

נקודה חשובה בנוגע לקיבוע משתנים היא העובדה שמהדר C# אינו מגביל גישה מטווח ההכרה הלא-בטוח למשתנה מקובע. לדוגמה, ניתן להשתמש במשתנה מקובע כערך ימני (r-value) לערך שמאלי (l-value) המוגדר בטווח הכרה רחב יותר מזה של בלוק קוד לא בטוח. דבר זה יכול לגרום ללא ספק לשימוש בערך לא בטוח מחוץ לבלוק זה. במקום שהמהדר יפלוט אזהרה או הודעת שגיאה, זו אחריות המפתח לשים לב כאשר משתמשים בערכים מקובעים כערכים ימניים.

# COM

חלק זה מיועד לכל אלה התוהים איך כל רכיבי COM שנכתבו במשך השנים יפעלו בדו-קיום עם סביבת הריצה של NET. כאן נראה איך רכיבי COM קלאסיים (כינוי די פוגע למען האמת) משתלבים בעולם של NET.

## עולם חדש ונועז

כפי שראית לאורך כל הספר, אין ספק שסביבת NET ושפת C# משתלבים יחד ליצירת אמצעי רב-עוצמה לבניית מערכות מבוססות-רכיבים. עם זאת, במשך השנים נצברה כמות עצומה של רכיבי COM הניתנים לניצול חוזר ואשר הושקע זמן רב ביצירתם. האם NET תביא לסופם של רכיבים אלה? האם הם יפעלו בדו-קיום עם סביבת הריצה המנוהלת של NET? ובכן, עבור אלה מבינינו המתפרנסים מ-COM, וגם אלה הסוגדים לביטוי "COM הוא טוב", יש חדשות טובות: יישומי NET framework יכולים לעבוד עם רכיבי COM קיימים. כפי שתראה בקרוב, רכיבי COM קלאסיים פועלים בשיתופיות עם זמן הריצה של NET באמצעות רובד שיתופיות (COM Interop) המטפל בכל "העבודה המלוכלכת" של העברת הודעות בין סביבת הריצה המנוהלת לבין רכיבי COM הפועלים בעולם הלא מנוהל.

# מתחילים

לפני שנעסוק ברובד השיתופיות נתחיל עם מספר הגדרות טכניות בקצרה ולאחר מכן נעבור לדוגמה מציאותית הממחישה מתי נרצה להשתמש ברכיב COM מיישום של NET. בהמשך אסביר בפירוט את הדוגמה ואיך תוכל לנצל את הידע שרכשת כאן ביישומים שאתה כותב.

בדוגמה זו, נניח שקיים רכיב COM בשם AirlineInfo אשר נכתב עם Visual C++ ו-ATL. כאן לא אעבור על כל השלבים הנדרשים לבניית רכיב זה מאחר וברצוני להתמקד בהיבטים של NET ו-#C. יחד עם זאת, אסביר את החלקים החשובים בקוד. תוכל למצוא את פרויקט Visual C++ במלואו בתקליטור המצורף לספר.

רכיב COM שבו נעסוק מתוכנן להפיק את פרטי הנחיתה עבור קו תעופה מסוים. לשם הפשטות, נניח שהרכיב מחזיר פרטים רק עבור הקו IC 5678 של Air Scooby ומחזיר שגיאה עבור כל קו אחר. מנגנון שגיאה זה הוכנס בכוונה בכדי להראות איך השגיאה שמעורר רכיב COM מופצת אחורנית ונתפסת על ידי לקוח NET.

הנה IDL עבור הרכיב:

```
interface IAirlineInfo : IDispatch
{
   [id(1), helpstring("method GetAirlineTiming")]
      HRESULT GetAirlineTiming([in] BSTR bstrAirline, [out,retval] BSTR* pBstrDetails);

   [propget, id(2), helpstring("property
      LocalTimeAtOrlando")] HRESULT
      LocalTimeAtOrlando([out, retval] BSTR
      *pVal);
};
```

אין כאן משהו מלהיב במיוחד, אפילו עבור רוב מפתחי COM המתחילים. זהו ממשק בשם IAirlineInfo עם שתי שיטות: GetAirlineTiming ו-LocalTimeAtOrlando. כעת נראה איך השיטה GetAirlineTiming מיושמת בפועל:

```
STDMETHODIMP CAirlineInfo::GetAirlineTiming(BSTR
   bstrAirline, BSTR *pBstrDetails)
{
   _bstr_t bstrQueryAirline(bstrAirline);
   if(NULL == pBstrDetails) return E_POINTER;

   if(_bstr_t("Air Scooby IC 5678") ==
      bstrQueryAirline)
```

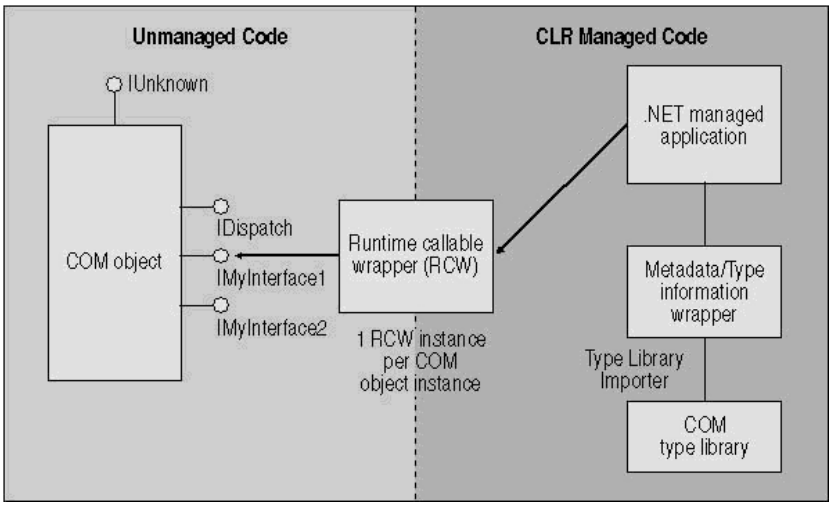
```
    {
        // Return the timing for this Airline.
        *pBstrDetails =
            _bstr_t(_T("16:45:00 - Will
            arrive at Terminal 3")).copy();
    }
        else
        {
            // Return an error message.
            return Error(LPCTSTR(_T("Not available" )),
                        __uuidof(AirlineInfo),
                        AIRLINE_NOT_FOUND);
    }
    return S_OK;
}
```

השיטה GetAirlineTiming לוקחת שני ארגומנטים. הראשון (bstrAirline) הוא BSTR המייצג את קו התעופה, והשני (pBstrDetails) הוא פרמטר output המחזיר את המידע על ההגעה (שעה מקומית ושער כניסה). בתוך השיטה בודקים אם הערך של bstrAirline שווה ל- "Air Scooby IC 5678". אם כן, מחזירים מידע כלשהו הכתוב בקידוד קשיח. אם אין זה הערך המצופה, קוראים לשיטת שגיאה המחזירה הודעה אודות העובדה שאנו תומכים בקו תעופה אחד בלבד.

לאחר סקירה בסיסית זו של הרכיב, הבה נסתכל על האופן שבו מייצרים מטא-נתונים מספריית הסוג של הרכיב (typelib) כדי שלקוח NET יוכל באמצעותם לשוחח עם הרכיב ולקרוא לשיטותיו.

## Com typelib-ו Metadata

יישום NET שצריך לשוחח עם רכיב COM אינו יכול לנצל בצורה ישירה את הפונקציונליות שהרכיב חושף. זאת מאחר וזמן הריצה של NET מתוכנן לעבוד עם רכיבים המכילים מטא-נתונים (ראה פרק 16), בעוד ש-COM מתוכנן לעבוד באמצעות רישום המערכת (Registry) וסדרה של שיטות תשאול המיושמות על ידי הרכיב. לפיכך, הדבר הראשון שיש לעשות בכדי לנצל את רכיב COM זה בעולם NET הוא ליצור עבורו מטא-נתונים. במקרה של רכיב COM, רובד מטא-נתונים זה משמש את זמן הריצה בקביעת מידע על סוגים. מידע זה משמש לאחר מכן בזמן ריצה ליצירת מה שנקרא RCW (runtime callable wrapper), (ראה תרשים 17-1). RCW מטפל בהפעלה הממשית של אובייקט COM ובהסדרת הפרמטרים הנדרשת כאשר יישום NET משוחח איתו. RCW מבצע עוד מספר רב של מטלות, כגון ניהול זהות אובייקט, משכי חיים של אובייקט ומטמון ממשק.




**תרשים 17-1** המרכיבים הבסיסיים של התפעוליות השיתופית בין NET ל-COM.

ניהול משך חיים הוא סוגיה קריטית מאחר ו-garbage collector של NET (GC) מעביר אובייקטים מכתובת לכתובת ומפנה אותם כאשר כבר אינם בשימוש. RCW משרת את המטרה של מימוש הפעולה המשותפת: הוא גורם ליישום NET "לחשוב" שהוא יוצר פעולות גומלין עם רכיב NET מנוהל, ונותן לרכיב COM במרחב הלא-מנוהל את הרושם שהוא נקרא על ידי לקוח COM מסורתי. היצירה וההתנהגות של RCW תלויים בכריכה מאוחרת או מוקדמת לאובייקט COM. מתחת לפני השטח, RCW מבצע את כל העבודה הקשה ודואג למיפוי כל הקריאות לשיטות של vtable (טבלת הפצה לפונקציות וירטואליות) לתוך רכיב COM הנמצא בעולם לא מנוהל. ביסודו של דבר הוא פועל כשגריר של רצון טוב בין העולם המנוהל לבין העולם IUnknown הלא-מנוהל.

הבה ניצור כעת את עטיפת המטא-נתונים עבור הרכיב AirlineInfo. לשם כך משתמשים בכלי הנקרא Type Library Importer (מייבא ספריית סוג - התוכנית tlbimp.exe). כלי עזר זה כלול ב-SDK של NET ומשמש לקריאת typelib של COM וליצירת עטיפת מטא-נתונים מתאימה, המכילה מידע על סוגים שזמן הריצה של NET יכול להבין. בכדי ליצור עטיפה זו עליך להתקין את יישומי ההדגמה מהתקליטור המצורף ולאתר את הרכיב AirlineInfo.

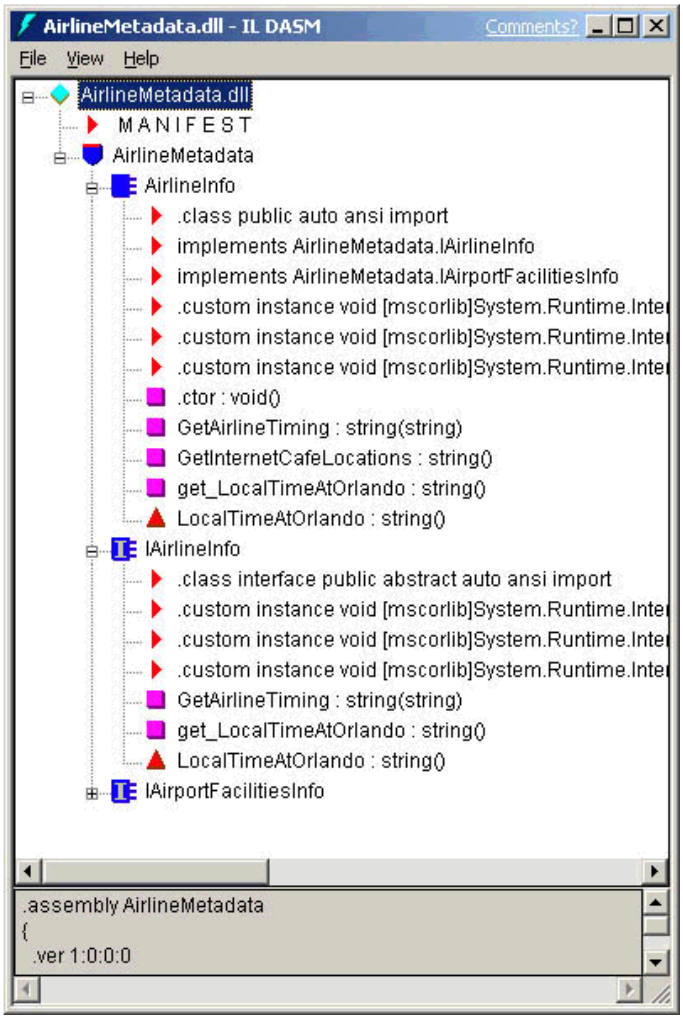
לאחר שעשית זאת, הקלד את השורה הבאה בשורת הפקודה:

```
TLBIMP AirlineInformation.tlb /out:AirlineMetadata.dll
```

הפקודה מורה ל-TLBIMP לקרוא את ספריית הסוג של AirlineInfo וליצור עטיפת מטא-נתונים תואמת בשם AirlineMetadata.dll. אם הכל יפעל כשורה תקבל את ההודעה הבאה:

```
TypeLib imported successfully to AirlineMetadata.dll
```

ובכן, מהו מידע הסוג הכלול במטא-נתונים שנוצרו, ואיך הוא נראה? כמפתחי COM, תמיד הערכנו את הכלי OleView.exe, בין השאר, בגלל יכולתו לאפשר לנו להציץ לתוך התכולה של typelib. למרבה המזל, בערכת הפיתוח של NET כלול כלי דומה: ILDASM (IL disassembler) (הוצג בפרק 2). כלי זה מאפשר לראות מטא-נתונים וקוד MSIL שנוצרו עבור מכללי קוד מנוהלים. כפי שלמדת בפרק 16, כל מכלל קוד מנוהל מכיל מטא-נתונים המתארים את עצמם, והכלי ILDASM שימושי מאוד כאשר צריך לחטט בנתונים אלה. תוכל להיווכח ביכולתו של כלי זה אם תפתח את AirlineMetadata.dll באמצעות ILDASM. אתה אמור לראות תוצאות דומות לאלו המוצגות בתרשים 17-2.




**תרשים 17-2** ILDASM הוא כלי נפלא לצפייה במטא-נתונים ובקוד MSIL עבור מכללים מנוהלים

במטא-נתונים שנוצרו תוכל להבחין בשיטה GetAirlineTiming הרשומה כחברה ציבורית של המחלקה AirlineInfo. בנוסף לכך קיים בנאי שנוצר עבור מחלקה זו. שים לב לכך שהפרמטרים של השיטה הוחלפו באופן אוטומטי בשווי-הערך המקבילים שלהם ב-NET. בדוגמה זו, BSTR הוחלף בפרמטר System.String. שים לב גם לכך שהפרמטר שנרשם כ-[out,retval] בשיטה GetAirlineTiming הומר לערך המוחזר בפועל של השיטה (מוחזר כ-System.String). בנוסף לכך, כל ערך כישלון של HRESULT המוחזר אחורה מאובייקט הרכיב - במקרה של שגיאה או לוגיקה עסקית כושלת - מעורר כחריג.

## Early Binding to COM Components

לאחר שייצרנו את המטא-נתונים הנדרשים על ידי לקוח NET, הבה נקרא לשיטה GetAirlineTiming ברכיב COM שלנו מתוך הלקוח. הנה יישום של לקוח #C היוצר אובייקט COM באמצעות המטא-נתונים שייצרנו מקודם וקורא לשיטה GetAirlineTiming. בדוגמה זו אנו משתמשים בכריכה מוקדמת. בקרוב אציג שתי דוגמאות נוספות המתארות כיצד לאתר סוגים בצורה דינאמית וכריכה מאוחרת.

```
using System;
using System.Runtime.InteropServices;
using System.Reflection;
using AIRLINEINFORMATIONLib ;

public class AirlineClient1App
{
   public static void Main()
   {
      ////////////////////////////////////////////////
      /// EARLY BINDING EXAMPLE
      ////////////////////////////////////////////////
      String strAirline = "Air Scooby IC 5678";
      String strFoodJunkieAirline = "Air Jughead TX 1234";
      try
      {
         AirlineInfo objAirlineInfo;

         // Create a new AirlineInfo Object.
         objAirlineInfo = new AirlineInfo();

         // Display the output after calling
         // the GetAirileTiming method.
         Console.WriteLine("Details for Airline {0} --> {1}",
         strAirline,objAirlineInfo.GetAirlineTiming(strAirline));
```

```
            // ERROR: The following will result in a thrown
            // exception!
            // Console.WriteLine("Details for Airline {0} --> {1}",
            // strFoodJunkieAirline,objAirlineInfo.GetAirlineTiming
            // (strFoodJunkieAirline));
        }
        catch(COMException e)
        {
            Console.WriteLine("Oops- We encountered an error " +
                              "for Airline {0}. The Error message " +
                              "is : {1}. The  Error code is {2}",
                              strFoodJunkieAirline ,
                              e.Message,e.ErrorCode);
        }
    }
}
```

זמן הריצה בונה RCW הממפה את השיטות והשדות של מחלקת מטא-נתונים לשיטות ולשדות אותם חושף הממשק המיושם על ידי הרכיב COM. מופע RCW אחד נוצר עבור כל מופע של אובייקט COM. זמן הריצה של NET דואג אך ורק לניהול משך החיים של RCW ופינויו. RCW הוא אשר דואג לתחזוקה של ספירת ייחוסים (reference counts) עבור רכיב COM אליו הוא ממופה, ובכך מבודד את זמן הריצה מהצורך לנהל ספירת ייחוסים עבור אובייקט הרכיב הממשי. כמוצג בתרשים 17-2, AirlineInfometadata מוגדר תחת מרחב שמות הנקרא AIRLINEINFORMATIONLib. לקוח NET רואה את שיטות הממשק כאילו היו חברי מחלקה של המחלקה AirlineInfo. כל שנדרש לעשות הוא ליצור מופע של המחלקה AirlineInfo באמצעות האופרטור new ולקרוא לשיטות המחלקה הציבוריות של האובייקט שנוצר. כאשר נקראת השיטה, RCW ממפה את הקריאה לשיטת COM מתאימה. RCW גם מטפל בכל סוגיות ההסדרה ומשך החיים של האובייקט. מנקודת מבט של הלקוח זה לא יותר מאשר ליצור אובייקט מנוהל טיפוסי וקריאה לאחת משיטות המחלקה הציבוריות שלו!

שים לב כי בכל פעם ששיטת COM מעוררת חריג, שגיאת COM זו נלכדת על ידי RCW. שגיאה זו מתורגמת לאחר מכן למחלקת COMException מתאימה (הנמצאת במרחב השמות System.Runtime.InteropServices). מובן שהרכיב עדיין צריך ליישם את הממשק ISupportErrorInfo בכדי שהפצת שגיאה זו תפעל וכך RCW יודע שהאובייקט מספק מידע שגיאה מורחב. לקוח NET יכול לתפוס את השגיאה באמצעות המנגנון הרגיל try-catch לטיפול בשגיאות, והוא מקבל גישה למספר השגיאה, תיאורה, מקור השגיאה ושאר הפרטים הזמינים לכל לקוח שפועל עם רכיב COM. בסעיף הבא נקדם דוגמה זו הלאה ונבחן דרכים אחרות של כריכה לרכיבי COM.

## Query Interface

כיצד פועל התסריט הקלאסי QueryInterface מנקודת מבט של לקוח NET הרוצה לגשת לממשק אחר המיושם על ידי אובייקט COM? בכדי לקבל QI עבור ממשק אחר כל שנדרש הוא להמיר (cast) את האובייקט הנוכחי לממשק האחר. כעת ניתן לקרוא לכל השיטות והמאפיינים של הממשק הרצוי.

RCW עושה כאן שוב את כל העבודה הקשה מתחת לפני השטח. הדבר מקביל במידת מה לאופן שבו זמן הריצה של Visual Basic מבודד מתכנתים מהצורך לכתוב קוד הקשור ל-QueryInterface - הוא פשוט עושה עבורך את QI כאשר אתה מגדיר אובייקט מסוג אחד כאובייקט מסוג משויך אחר.

הדוגמה הבאה ממחישה עד כמה זה פשוט. נניח שרוצים לקרוא לשיטות בממשק IAirportFacilities, שהוא ממשק אחר אותו מיישם רכיב COM מהדוגמאות הקודמות. לשם כך ממירים את האובייקט AirlineInfo לממשק IAirportFacilities. כעת ניתן לקרוא לכל השיטות שהן חלק מהממשק IAirportFacilities. ואולם לפני ההמרה אולי נרצה לבדוק אם מופע האובייקט הנוכחי תומך בממשק המבוקש או מיישם אותו. עושים זאת באמצעות השיטה IsInstanceOf של המחלקה System.Type. אם השיטה מחזירה true אזי QI הצליח ואנו יכולים לבצע את ההמרה ללא חשש. אם מנסים להמיר את האובייקט לממשק שרירותי כלשהו שאינו נתמך על ידי האובייקט, נזרק החריג System.InvalidCastException. בדרך זו RCW מבטיח שנבצע המרה רק לממשקים המיושמים על ידי אובייקט הרכיב. כך נראה כל זה בקוד:

```
using System;
using System.Runtime.InteropServices;
using System.Reflection;
using AIRLINEINFORMATIONLib;

public class AirlineClient2App
{
   public static void Main()
   {
      ////////////////////////////////////////////////
      /// QUERY INTERFACE/ RT type Checking
      ////////////////////////////////////////////////
      try
      {
         AirlineInfo objAirlineInfo;
         IAirportFacilitiesInfo objFacilitiesInfo;
```

```
        // Create a new AirlineInfo object.
        objAirlineInfo = new AirlineInfo();

        // Invoke the GetAirlineTiming method.
        String strDetails = objAirlineInfo.GetAirlineTiming
                                (strAirline);

        // QI for the IAirportFacilitiesInfo interface.
        objFacilitiesInfo =
              (IAirportFacilitiesInfo)objAirlineInfo;

        //Invoke a method on the IAirportFacilitiesInfo
        //interface.
        Console.WriteLine("{0}",
        objFacilitiesInfo.GetInternetCafeLocations());
      }
      catch(InvalidCastException eCast)
      {
        Console.WriteLine("We got an InvalidCast Exception " +
                    "- Message is {0}",eCast.Message);
      }
    }
}
```

## Late Binding to COM Components

שתי הדוגמאות שראינו עד כה - AirlineClient1App ו-AirlineClient2App - כורכות שתיהן את לקוח NET עם אובייקט הרכיב באמצעות מטא-נתונים. למרות שורת יתרונות הפתיחה של כריכה מוקדמת - בדיקת סוגים חזקה בזמן הידור, יכולות השלמה אוטומטית (ממידע על סוג) של כלי הפיתוח (כגון Visual Studio NET) וכמובן, ביצועים טובים יותר - יש מקרים בהם אין ברשותך מטא-נתונים של זמן הידור עבור רכיב COM ולכן עליך לכרוך אותו עם כריכה מאוחרת (late binding). לדוגמה, אם הרכיב שבו אתה מנסה להשתמש מכיל רק את הממשק dispinterface, אתה מוגבל מאוד לשימוש ברכיב עם כריכה מאוחרת.

כריכה מאוחרת של רכיב COM מנצלת את מנגנון השיקוף עליו למדת בפרק 16. בכדי לכרוך רכיב COM בדרך זו יש לדעת את ProgID של הרכיב. זאת מכיון שהשיטה הסטטית CreateInstance של המחלקה System.Activator דורשת אובייקט Type. אולם, באמצעות ProgID של הרכיב ניתן לקרוא לשיטה GetTypeFromProgID של המחלקה System.Type, המחזירה אובייקט Type חוקי של NET שניתן להשתמש בו לאחר מכן בקריאה ל-System.Activator.CreateInstance. מרגע זה ניתן לקרוא לכל אחד מהשיטות או המאפיינים הנתמכים על ידי ממשק המחדל של הרכיב, באמצעות שיטת המופע System.Type.InvokeMember של האובייקט Type המוחזר מ-GetTypeFromProgID.

כל שנדרש הוא לדעת את שם השיטה או המאפיין, ואת המידע על הפרמטר שהקריאה לשיטה מקבלת. כאשר קוראים לשיטה של רכיב כריכה מאוחרת, מעבירים פרמטרים על ידי אריזתם בתוך מערך System.Object כללי והעברתם לשיטה. צריך גם להגדיר את דגלי הכריכה (binding flags) המתאימים, תלוי אם קוראים לשיטה או קוראים/כותבים ערך של מאפיין.

בקוד הבא רואים שנדרשת מעט יותר עבודה מאשר עם כריכה מוקדמת. עם זאת, במקרים בהם כריכה מאוחרת היא הברירה היחידה, ודאי תהיה מרוצה מיכולת זו.

```
using System;
using System.Runtime.InteropServices;
using System.Reflection;
using AIRLINEINFORMATIONLib;

public class AirlineClient3App
{
   public static void Main()
   {
      ////////////////////////////////////////////////
      /// LATE BINDING
      ////////////////////////////////////////////////
      try
      {
         object objAirlineLateBound;
         Type objTypeAirline;

         object[] arrayInputParams= { "Air Scooby IC 5678" };

         objTypeAirline = Type.GetTypeFromProgID
               ("AirlineInformation.AirlineInfo");

         objAirlineLateBound = Activator.CreateInstance
                           (objTypeAirline);

         String str = (String)objTypeAirline.InvokeMember
                           ("GetAirlineTiming",
                           BindingFlags.Default |
                           BindingFlags.InvokeMethod,
                           null, objAirlineLateBound,
                           arrayInputParams);
```

```
        Console.WriteLine("{0}",str);

        String strTime = (String)objTypeAirline.InvokeMember
                            ("LocalTimeAtOrlando",
                            BindingFlags.Default |
                            BindingFlags.GetProperty,
                            null, objAirlineLateBound,
                            new object [] {});

        Console.WriteLine ("Hi there !. The Local Time in " +
                        "Orlando,Florida is: {0}", strTime);
    }
    catch(COMException e)
    {
        Console.WriteLine("Oops- We encountered an error " +
                    "for Airline {0}. The Error message " +
                    "is : {1}. The Error code is {2}",
                    strFoodJunkieAirline,
                    e.Message,e.ErrorCode);
    }
  }
}
```

## COM Threading Models

למרבית המפתחים העושים את צעדי התכנות הראשונים ב-COM יש מושג קלוש, או שאין כלל, על מודל המטלה ועל COM apartments. רק לאחר צבירת ניסיון משמעותי הם מבינים שלמודל המטלה החופשית בו הם השתמשו יש מחיר כבד מבחינת ביצועים כשמשתמשים במטלת לקוח מהסוג single-threaded apartment (STA) ליצירת אובייקט multithreaded apartment (MTA). בנוסף לכך, מתכנתים מתחילים ב-COM אינם מודעים לעיתים קרובות לסוגיה של בטיחות מטלה (thread safety) ולסכנה הקרבה ובאה עליהם כאשר מטלות הפועלות בו-זמנית ניגשות לרכיבי COM שלהם.

לפני שמטלה יכולה לגשת לתוך אובייקט COM, עליה להצהיר על כניסתה ל-apartment מסוג STA או MTA. מטלות לקוח מסוג STA קוראות לפונקציה CoInitialize(NULL) או CoInitializeEx(0, COINIT_APARTMENTTHREADED) בכדי להיכנס ל-STA, ומטלות MTA קוראות לפונקציה CoInitializeEx(0, COINIT_MULTITHREADED) בכדי להיכנס ל-MTA. באופן דומה, בעולם המנוהל של NET קיימת האפשרות שהמטלה הקוראת במרחב המנוהל תצהיר על כניסתה ל-apartment. על פי ברירת המחדל, המטלה הקוראת ביישום מנוהל בוחרת לחיות ב-MTA. זה כאילו שהמטלה הקוראת מאתחלת את עצמה עם CoInitializeEx(0, COINIT_MULTITHREADED). עם זאת, חשוב על מחיר התקורה והביצועים אם המטלה היתה קוראת לרכיב STA קלאסי שתוכנן להיות apartment-threaded. incompatible apartments גורמות לתקורה של צמד proxy/stub נוסף, וזה בא בוודאות על חשבון הביצועים.

לשם כך ניתן לדרוס את בחירת המחדל של apartment עבור מטלה מנוהלת ביישום NET באמצעות המאפיין ApartmentState של המחלקה System.Threading.Thread. המאפיין ApartmentState לוקח אחד מהקבועים הממוספרים הבאים:

- **MTA** - Multithreaded apartment
- **STA** - Single-threaded apartment
- **Unknown** - שווה-ערך להתנהגות ברירת המחדל MTA.

צריך להגדיר את המאפיין ApartmentState עבור המטלה הקוראת עוד לפני שקוראים לאובייקט COM. שים לב שלא ניתן לשנות את ApartmentState מרגע שנוצר אובייקט של הרכיב. לפיכך, יש היגיון בהגדרת ApartmentState של המטלה מוקדם ככל האפשר בקוד. הקוד הבא ממחיש את הדרך לעשות זאת:

```
// Set the client thread ApartmentState to enter an STA.
Thread.CurrentThread.ApartmentState =
   ApartmentState.STA;

// Create our COM object through the Interop.
MySTA objSTA = new MySTA();
objSTA.MyMethod()
```

# סיכום

פיסת המידע האחרונה עוסקת באופן שבו המנגנונים השונים לעבודה עם קוד קיים - PInvoke, קוד לא בטוח ורובד COM Interop - משתלבים בסכמת המרכיבים הכוללת של NET. בפרק זה למדת את הדברים הבאים:

- בנוגע לשימוש בקריאות לפונקציות דמויות-C, למדת איך להשתמש ב-Pinvoke יחד עם מספר תכונות אשר הופכות סוגיות של הסדרת סוגי נתונים שונים, כולל נתונים מותאמים, לקלות יותר.
- בנוגע לקוד לא בטוח, למדת איך לוותר על היתרונות של קוד מנוהל בתוך יישום #C לטובת מצבים שבהם אתה זקוק לשליטה נוספת. תסריטים אלה כוללים מקרים שבהם אתה צריך לטפל בזיכרון באופן ידני למטרות יעילות, או כאשר אתה מעביר בלוקים של קוד לתוך יישום #C ופשוט אינך מוכן עדיין להמיר קוד זה לקוד מנוהל.
- בנוגע ל-COM, למדת איך לחשוף רכיבי COM קלאסיים ליישומי NET ואיך רובד השיתופיות COM Interop מאפשר לך ניצול חוזר באופן חלק של רכיבי COM מקוד מנוהל. לאחר מכן עברת בקצרה על דרכים להפעיל רכיב COM באמצעות כריכה מוקדמת וכריכה מאוחרת לצד דרכים לבצע בדיקת סוגים בזמן ריצה. לסיום, ראית איך מטלות מנוהלות מצהירות על כניסתן ל-apartments כאשר הן מפעילות רכיבי COM.

בנקודה זו, כמפתח המשתמש באחד ממנגנונים אלה לטיפול בקוד לא מנוהל, אתה עשוי לתהות האם הגיוני להמשיך ולהשתמש במתודולוגיות אלו או לעבור ישירות לתוך עולם NET ולכתוב רכיבים וקוד מנוהלים עבור הלוגיקה העסקית שלך באמצעות שפה של NET כגון #C. התשובה תלויה במצבך הנוכחי.

אם ברשותך כמות עצומה של קוד קיים - בין אם אלו פונקציות דמויות-C בקבצי DLL, קוד המטפל ישירות בזיכרון, רכיבי COM, או צירוף כלשהו של השלושה - למעשה לא תהיה מסוגל להמיר את כל הקוד שלך בין לילה. במקרה זה, הגיוני לנצל את מנגנוני NET השונים לעבודה עם קוד קיים. יחד עם זאת, אם אתה כותב לוגיקה עסקית חדשה מאפס, אני ממליץ לך מכל הלב לכתוב את הקוד כרכיבים מנוהלים תוך שימוש בשפה כמו #C. בדרך זו תוכל להיפטר מהמחיר הבלתי נמנע של ירידה בביצועים כאשר נעים בין התחום המנוהל לתחום הלא מנוהל.

# פרק 18 Assemblies

פרק זה מתאר את יתרונות השימוש העיקריים במכללי קוד, כולל אריזה (packaging) והבחנת גרסאות (versioning) של רכיבי NET. כאן נלמד איך ליצור מכללי קובץ אחד ומכללי קבצים מרובים באמצעות הכלי Assembly Generation (al.exe), איך ליצור מכללים משותפים באמצעות הכלי Strong Name (sn.exe), איך לעיין במטמון הגלובלי של המכלל באמצעות ההרחבה Assembly Cache Viewer (shfusion.dll), ואיך לטפל במטמון המכלל בעזרת הכלי Global Assembly Cache (gacutil.exe). לסיום נראה מספר דוגמאות של הבחנת גרסאות ואיך מכללים ומדיניות ההבחנה בין גרסאות של NET עוזרים להימנע ממה שנקרא בעגת המתכנתים "DLL hell".

## סקירה כללית

בפרק 16 תוארו מכללים כקבצים פיסיים המורכבים מקובץ הפעלה נייד (PE) אחד או יותר הנוצרים על ידי מהדר NET. בהקשר של אותו פרק, ההגדרה התקבלה על הדעת. עם זאת, מכללים הם יותר מורכבים מהגדרה זו. הנה הגדרה מושלמת יותר: **מכלל קוד** הוא אריזה המכילה **מצהר** (manifest), מודול אחד או יותר, ובאופן אופציונלי, משאב אחד או יותר. באמצעות מכללים ניתן לקבץ יחידות פונקציונליות בעלות קירבה סמנטית בקובץ אחד למטרות של הטמעה, אפיון גירסה ותחזוקה.

כל קבצי PE המשתמשים בזמן הריצה של NET מורכבים ממכלל או מקבוצה של מכללי קוד. כאשר מהדרים יישום באמצעות מהדר C#, יוצרים למעשה מכלל קוד. עובדה זו עשויה שלא להיות מובנת אלא אם מנסים באופן מפורש להציב מודולים מרובים במכלל אחד או מנצלים תכונה ייחודית מסוימת של מכלל כגון אפיון גירסה.

עם זאת, חשוב להבין שבכל זמן שבונים EXE או DLL (באמצעות המתג t:library/), יוצרים מכלל עם מצהר המתאר את המכלל עבור זמן הריצה של NET. בנוסף לכך, ניתן ליצור מודול (באמצעות המתג t:module/) שהוא DLL ממשי (עם סיומת netmodule) ללא מצהר. במילים אחרות, למרות שמבחינה לוגית זהו עדיין DLL, הוא אינו שייך למכלל וחייבים לצרפו למכלל באמצעות המתג addmodule/ בזמן ההידור או באמצעות הכלי Assembly Creation. תלמד איך עושים זאת מאוחר יותר בסעיף "Building Assemblies".

## Manifest Data

ניתן לאחסן מצהר של מכלל בדרכים שונות. אם בכוונתך להדר יישום עצמאי או DLL, יש להכליל את המצהר בתוך PE המתקבל. מכלל זה ידוע בשם מכלל קובץ אחד (single-file assembly). ניתן ליצור גם מכלל קבצים מרובים (multifile assembly), כאשר המצהר קיים כיישות עצמאית בתוך המכלל או מצורף לאחד המודולים בתוך המכלל.

ההגדרה של מכלל גם תלויה במידה רבה באופן השימוש בו. מנקודת המבט של הלקוח, מכלל הוא אוסף של מודולים, סוגים מיוצאים ומשאבים (אופציונלי), בעלי שם וגירסה. מנקודת המבט של היוצר, מכלל הוא אמצעי לארוז מודולים, סוגים ומשאבים בעלי קירבה, ולייצא רק את מה שהלקוח צריך. עם זאת, המצהר הוא זה אשר מספק את רמת ההפרדה בין פרטי המימוש של המכלל לבין מה שהלקוח אמור להשתמש בו. הנה פירוט המידע הנשמר במצהר של מכלל קוד:

- **שם המכלל (assembly name)** - השם הטקסטואלי של המכלל.
- **מידע אפיון גירסה (versioning information)** - המחרוזת מורכבת מארבעה מקטעים נפרדים היוצרים יחד את מספר הגירסה. הם כוללים מספר גירסה ראשי ומשני וגם מספר תיקון (revision) ובנייה (built).
- **שם משותף (shared name) ו-hash מכלל מסומן (signed assembly hash) (אופציונליים)** - מידע זה קשור להטמעה של מכללים ונסקר מאוחר יותר בסעיף "Assembly Deployment".
- **קבצים (files)** - רשימה זו כוללת את כל הקבצים הנמצאים במכלל.
- **ייחוסים למכללים חיצוניים (referenced assemblies)** - רשימת כל המכללים החיצוניים שמתייחסים אליהם ישירות מתוך המכלל של המצהר.
- **סוגים (types)** - רשימת כל הסוגים במכלל עם מיפוי למודול המכיל את הסוג. מידע זה עוזר לדוגמת השיקוף בפרק 16 (המבצעת איטרציה על כל הסוגים במכלל) להתבצע כה מהר.

- **הרשאות אבטחה (security permissions)** - רשימת כל הרשאות האבטחה הנדחות באופן מפורש על ידי המכלל.
- **תכונות מותאמות (custom attributes)** - פרק 8 תיאר איך ליצור תכונות מותאמות אישית. כמו סוגים, תכונות מותאמות מאוחסנות במצהר של המכלל לגישה מהירה במהלך שיקוף.
- **מידע מוצר (product information)** - מידע זה כולל את שם החברה, סימן מסחרי וערכים של זכויות יוצרים.

מכללים מספקים למפתח מספר יתרונות, כולל **אריזה** (packaging), **הטמעה** (deployment) **והבחנת גירסה** (versioning).

## Assembly Packing

היתרון ביכולת לארוז מודולים מרובים בקובץ פיסי אחד הוא שיפור בביצועים. כאשר יוצרים יישום ומטמיעים אותו באמצעות מכלל קבצים מרובים, זמן הריצה של NET צריך לטעון רק את המודולים הנחוצים. לכך יש השפעה בהפחתת מערך העבודה של היישום.

## Assembly Deployment

יחידת ההטמעה המינימלית ב-NET היא המכלל. כפי שציינתי מקודם, ניתן ליצור מודול עם סיומת netmodule באמצעות המתג t:module/, אך חייבים להכלילו במכלל אם רוצים להטמיע אותו. בנוסף לכך, למרות הפיתוי לומר שמכללים הם אמצעי להטמעת יישום, אין זה נכון מבחינה טכנית. מדויק יותר לראות מכללים ב-NET כצורה של הטמעת מחלקות (בדומה מאוד לקבצי DLL ב-Win32), שבה יישום אחד יכול להיבנות ממכללים רבים.

מאחר ומכללים מתארים את עצמם, הדרך הקלה ביותר להטמיע אותם היא להעתיק את המכלל לתיקיית היעד הרצויה. לאחר מכן, כאשר מנסים לבצע יישום הכלול במכלל, המצהר מכוון את זמן הריצה של NET למודולים הכלולים במכלל. בנוסף, המכלל גם מכיל ייחוסים למכללים חיצוניים כלשהם הנחוצים ליישום.

האמצעי השכיח ביותר להטמעה הוא **מכללים פרטיים** (private assemblies) - כלומר, מכללים המועתקים לתיקיה ואינם משותפים. מכלל מוגדר באופן אוטומטי פרטי כברירת מחדל אלא אם מגדירים מפורשות את המכלל כ**מכלל משותף** (shared assembly). שיתוף מכללים דורש מעט יותר עבודה ונושא זה נסקר מאוחר יותר בסעיף "Creating Shared Assemblies".

## Assembly Versioning

יתרון גדול אחר לשימוש במכללים הוא הבחנה מובנית בין גרסאות - ליתר דיוק, הקץ ל- "DLL hell". ביטוי זה מתייחס למצב שבו יישום אחד דורס DLL המשותף ליישום אחר, בדרך כלל עם גירסה ישנה יותר של אותו DLL, דבר המוביל בסופו של דבר לקריסת היישום הראשון. למרות שפורמט קובץ המשאבים של Win32 מאפשר אפיון גרסת סוג משאב, מערכת ההפעלה אינה כופה שום כללי הבחנה בין גרסאות כדי שיישומים ימשיכו לתפקד. אחריות זו מוטלת אך ורק על כותבי היישום.

כאמצעי לטיפול בסוגיה זו, המצהר כולל מידע גירסה עבור המכלל וגם רשימת כל המכללים שיש אליהם ייחוס ואת מידע הגירסה עבורם. הודות לארכיטקטורה זו, זמן הריצה של NET יכול להבטיח שמדיניות ההבחנה בין גרסאות תישמר ויישומים ימשיכו לתפקד גם כאשר גרסאות חדשות, לא מתאימות, של קבצי DLL משותפים מותקנות במערכת. היות והבחנת גרסאות היא אחד היתרונות הגדולים ביותר של מכללים, היא נסקרת לעומק, כולל מספר דוגמאות, בסעיף "Versioning Assemblies".

# Building Assemblies

אם יוצרים DLL עם המתג t:library/, לא ניתן לצרף אותו למכלל אחר. זאת מאחר והמהדר יוצר באופן אוטומטי מצהר עבור DLL, ולכן DLL עצמו הוא מכלל. הדוגמה הבאה ממחישה זאת. כאן יש DLL (Module1Server.cs) המכיל סוג דמה בשם Module1Server.

```
// Module1Server.cs
// build with the following command line switches
//     csc /t:library Module1Server.cs
public class Module1Server
{
}
```

לאחר מכן פונה אל DLL זה קוד הלקוח (Module1Client.cs):

```
// Module1ClientApp.cs
// build with the following command line switches
//     csc Module1ClientApp.cs /r:Module1Server.dll
using System;
using System.Diagnostics;
using System.Reflection;
```

```
class Module1ClientApp
{
   public static void Main()
   {
      Assembly DLLAssembly = Assembly.GetAssembly(typeof(Module1Server));
      Console.WriteLine("Module1Server.dll Assembly Information");
      Console.WriteLine("\t" + DLLAssembly);

      Process p = Process.GetCurrentProcess();
      string AssemblyName = p.ProcessName + ".exe";
      Assembly ThisAssembly = Assembly.LoadFrom(AssemblyName);
      Console.WriteLine("Module1Client.exe Assembly Information");
      Console.WriteLine("\t" + ThisAssembly);
   }
}
```

עתה נניח שבונים שני מודולים אלה באמצעות המתגים הבאים:

```
csc /t:library Module1Server.cs
csc Module1ClientApp.cs /r:Module1Server.dll
```

אם נבצע את הקוד בנקודה זו יתקבל הפלט הבא, המוכיח כי EXE וגם DLL נמצאים במכללים נפרדים משל עצמם:

```
Module1Server.dll Assembly Information
   Module1Server, Version=0.0.0.0, Culture=neutral, PublicKeyToken=null
Module1Client.dll Assembly Information
   Module1Client, Version=0.0.0.0, Culture=neutral, PublicKeyToken=null
```

למעשה, אם היינו משנים את מציין הגישה של המחלקה Module1Server מ-public ל-internal, לא ניתן היה להדר את הקוד מאחר ועל פי ההגדרה internal מציין שהסוג זמין אך ורק לקוד אחר הנמצא באותו מכלל

## Multiple Modules

ניתן להציב את שני המודולים מהדוגמה הקודמת במכלל אחד בשתי דרכים. הדרך הראשונה היא לשנות את המתגים עבור המהדר. הנה דוגמה:

```
// Module2Server.cs
// build with the following command line switches
//     csc /t:module Module2Server.cs
internal class Module2Server
{
}
```

שים לב שכעת ניתן להשתמש במציין internal כך שהמחלקה זמינה רק לקוד בתוך המכלל.

```
// Module2ClientApp.cs
// build with the following command line switches
//     csc /addmodule:Module2Server.netmodule Module2ClientApp.cs
using System;
using System.Diagnostics;
using System.Reflection;

class Module2ClientApp
{
   public static void Main()
   {
      Assembly DLLAssembly =
         Assembly.GetAssembly(typeof(Module2Server));
      Console.WriteLine("Module1Server.dll Assembly Information");
      Console.WriteLine("\t" + DLLAssembly);

      Process p = Process.GetCurrentProcess();
      string AssemblyName = p.ProcessName + ".exe";
      Assembly ThisAssembly = Assembly.LoadFrom(AssemblyName);
      Console.WriteLine("Module1Client.dll Assembly Information");
      Console.WriteLine("\t" + ThisAssembly);
   }
}
```

שים לב לאופן הבנייה של Module2Server.cs ו-Module2Client.exe:

```
csc /t:module Module2Server.cs
csc /addmodule:Module2Server.netmodule Module2Client.cs
```

תחילה יש להסיר את המתג r/ מאחר ומתג זה משמש להפניה למכללים וכעת שני המודולים שוכנים באותו מכלל. לאחר מכן יש להכניס את המתג addmodule/, המורה למהדר אילו מודולים יש להוסיף למכלל שנוצר.

הנה התוצאות המתקבלות לאחר הידור וביצוע היישום:

```
Module1Server.dll Assembly Information
   Module2Client, Version=0.0.0.0, Culture=neutral, PublicKeyToken=null
Module1Client.dll Assembly Information
   Module2Client, Version=0.0.0.0, Culture=neutral, PublicKeyToken=null
```

דרך אחרת ליצור מכלל היא בעזרת הכלי Assembly Generation. כלי זה מקבל קלט של קובץ אחד או יותר, היכולים להיות מודולי NET (מכילים קוד MSIL) או קבצי משאבים וקבצי תמונות. הפלט הוא קובץ המכיל מצהר של מכלל.

דוגמה לשימוש בכלי זה היא כאשר יש מספר קבצי DLL שרוצים להפיצם ולהעניק להם גירסה כיחידה אחת. בהנחה שאלה נקראים A.DLL, B.DLL ו-C.DLL, משתמשים ביישום al.exe ליצירת המכלל המורכב, הנה כך:

```
al /out:COMPOSITE.DLL A.DLL B.DLL C.DLL
```

# Creating Shared Assemblies

יוצרים מכללים משותפים כאשר מכלל אמור לשמש יישומים מרובים והבחנת הגרסאות חשובה (בחלק הבא נעסוק בהבחנת גרסאות). בכדי לשתף מכלל חייבים ליצור **שם משותף** (shared name), הנקרא גם **שם חזק** (strong name) עבור המכלל באמצעות הכלי Strong Name המצורף ל-SDK של NET. להלן ארבעת היתרונות החשובים הנובעים מהשימוש ב-strong names:

- strong name הוא מנגנון NET ליצירת שם גלובלי ייחודי.
- היות וצמד המפתחות שנוצר (יוסבר בקרוב) כולל חתימה, ניתן לקבוע אם הוא שונה לאחר יצירתו המקורית.
- strong names מבטיחים שצד שלישי לא יוכל לשחרר גירסה עוקבת של המכלל שבנית, היות ובגלל החתימות אין לצד שלישי את המפתח הפרטי שלך.
- כאשר NET טוענת מכלל, זמן הריצה יכול לוודא שהמכלל הגיע מאותו מוציא לאור שהקוד הקורא מצפה לו.

השלב הראשון ביצירת strong name הוא להשתמש בכלי Strong Name ליצירת קובץ מפתח עבור המכלל. עושים זאת על ידי ציון המתג k- עם שם קובץ הפלט שיכיל את המפתח. לדוגמה, ניצור קובץ מפתח בשם InsideCSharp.key באופן הבא:

```
sn -k InsideCSharp.key
```

אם מבצעים פקודה זו אמורים לקבל את הודעת האישור הבאה:

```
Key pair written to InsideCSharp.key
```

כעת נוסיף את התכונה assembly:AssemblyKeyFile לקובץ המקור. כאן יצרתי מערך קבצים פשוט אחר בכדי להמחיש זאת:

```
// Module3Server.cs
// build with the following command line switches
//     csc /t:module Module3Server.cs
internal class Module3Server
{
}
```

```
// Module3ClientApp.cs
// build with the following command line switches
//     csc /addmodule:Module3Server.netmodule Module3ClientApp.cs
using System;
using System.Diagnostics;
using System.Reflection;

[assembly:AssemblyKeyFile("InsideCSharp.key")]

class Module3ClientApp
{
   public static void Main()
   {
      Assembly DLLAssembly =
         Assembly.GetAssembly(typeof(Module3Server));
      Console.WriteLine("Module1Server.dll Assembly Information");
      Console.WriteLine("\t" + DLLAssembly);

      Process p = Process.GetCurrentProcess();
      string AssemblyName = p.ProcessName + ".exe";
      Assembly ThisAssembly = Assembly.LoadFrom(AssemblyName);
      Console.WriteLine("Module1Client.dll Assembly Information");
      Console.WriteLine("\t" + ThisAssembly);
   }
}
```

כפי שרואים, בנאי התכונה assembly:AssemblyKeyFile לוקח את שם קובץ המפתח שנוצר עם הכלי Strong Name שהוא האמצעי להגדרת צמד מפתחות המשמש למתן strong name עבור המכלל. חשוב להבין שתכונה זו היא תכונה ברמת המכלל. לפיכך, מבחינה טכנית ניתן להציבה בכל קובץ במכלל והיא אינה מצורפת למחלקה ספציפית. עם זאת, מקובל למקם תכונה זו מייד לאחר משפטי using ולפני הגדרות המחלקות.

כאשר תבצע את היישום, שים לב לערך PublicKeyToken של המכלל. ערך זה היה null בשתי הדוגמאות הקודמות מאחר ומכללים אלה נחשבו פרטיים. אולם כעת המכלל הוגדר כמכלל משותף ולכן יש לו סימן מפתח ציבורי (public key token) משויך.

```
Module3Server.dll Assembly Information
   Module3Client, Version=0.0.0.0, Culture=neutral,
   PublicKeyToken=6ed7cef0c0065911
Module3Client.dll Assembly Information
   Module3Client, Version=0.0.0.0, Culture=neutral,
   PublicKeyToken=6ed7cef0c0065911
```

אובייקט Assembly שיצר מופע עבור דוגמה זו הוא משותף. עם זאת, איך יודעים אילו מכללים במערכת NET שלנו הם משותפים? התשובה היא מטמון המכלל הגלובלי. בסעיף הבא אסקור חלק זה של NET ואסביר את תפקידו במכללים משותפים.

# Global Assembly Cache

לכל התקנה של NET יש מטמון קוד הנקרא **מטמון מכלל גלובלי** (global assembly cache). תחום זה משרת שלוש מטרות עיקריות:

- הוא משמש לאחסון קוד שנטען מהאינטרנט או משרתים אחרים (שרתי http וגם שרתי קבצים). שים לב כי הקוד שנטען עבור יישום מסוים מאוחסן בחלק הפרטי של המטמון ובכך הוא אינו זמין לאחרים.
- הוא משמש כמחסן נתונים עבור רכיבים משותפים ליישומי NET מרובים. מכללים המותקנים לתוך המטמון באמצעות הכלי Global Assembly Cache מאוחסנים בחלק הגלובלי של המטמון והם זמינים לכל היישומים במכונה.
- שאלה אותה אני שומע לעיתים קרובות היא: "היכן נשמר קוד המכונה (jitted code) שנוצר עבור מכלל הקוד כך שקוד #C שלי מהודר לקוד מכונה רק בפעם הראשונה שבה הוא מתבצע?" כעת אתה יודע את התשובה: גרסאות קוד מכונה של מכללים שעברו הידור בזמן התקנה מאוחסנות במטמון.

## עיון במטמון

הבה נעיין במטמון בכדי לראות את המכללים המותקנים והמשותפים הנוכחיים. באמצעות סייר Windows פתח את התיקיה c:\winnt\assembly. בכדי לתמוך בעיון במידע מתאים אודות מכללים, NET מציגה הרחבת מעטפת הנקראת Assembly Cache Viewer (shfusion.dll). כלי זה מאפשר לך לצפות במידע על מכלל, כגון מספר גירסה, סימן מפתח ציבורי, ואפילו תוכל לדעת אם המכלל עבר הידור מוקדם (prejitted).

אמצעי אחר לצפייה במטמון הוא הכלי Global Assembly Cache. כלי זה מאפשר לבצע מספר משימות בסיסיות על ידי הגדרת אחד (ורק אחד) ממתגי שורת הפקודה הבאים.

- **i -** דגל זה מתקין מכלל במטמון המכלל הגלובלי. לדוגמה:

```
gacutil -i HelloWorld.DLL
```

בקרוב תלמד איך להוסיף את המכלל Module3Client למטמון באמצעות מתג זה.

- **u -** דגל זה מסיר התקנה של מכלל, כולל כל מידע על גירסה, ממטמון המכלל הגלובלי. אם לא מציינים את מידע הגירסה, מוסרת ההתקנה של כל המכללים עם השם המסוים. לפיכך, הדוגמה הראשונה להלן מסירה את ההתקנה של כל המכללים הנקראים HelloWorld ללא תלות במספר הגירסה, והדוגמה השנייה מסירה את ההתקנה של גירסה מסוימת:

```
gacutil -u HelloWorld
gacutil -u HelloWorld, ver=1,0,0,0
```

- **l -** דגל זה מפרט את התכולה של מטמון המכלל הגלובלי, כולל שם המכלל, מספר הגירסה שלו, מיקומו ושמו המשותף.

במספר גרסאות בטא קודמות של NET שמתי לב לבעיה, הרחבת המעטפת לא פעלה בתיקיה c:\winnt\assembly. הסיבה לכך נבעה מהעובדה שההרחבה shfusion.dll לא נרשמה באופן תקין. אם תופעה זו מתרחשת במערכת שלך, פתח את חלון שורת הפקודה והכנס את השורה הבאה מתוך התיקיה c:\winnt\Microsoft.net\framework\vXXX, כאשר XXX מייצג את מספר הגירסה של מסגרת NET אותה אתה מפעיל. מאחר ואני משתמש בגרסת בטא, ברור ששם התיקיה ישתנה לפני ההפצה של NET. חפש את הקובץ shfusion.dll והשתמש בתיקיה זו. כאן השתמשתי בתיקיה המייצגת את מספר הבנייה (built) הנוכחי של NET במערכת שלי:

```
c:\winnt\microsoft.net\framework\v1.0.2615>regsvr32 shfusion.dll
```

לאחר שיצרנו קובץ מפתח ציבורי והקצינו אותו למכלל, הבה נוסיף מכלל זה למטמון. לשם כך, מקלידים את השורה הבאה בשורת הפקודה:

```
gacutil -i Module3ClientApp.exe
```

אם הכל פועל כשורה, אמורים לקבל את הודעת האישור הבאה:

```
Assembly successfully added to the cache
```

בנקודה זו ניתן להשתמש בפקודה gacutil-l בכדי לעיין במכללים הרשומים במטמון ולמצוא את Module3Client, או שניתן להשתמש בכלי Assembly Cache Viewer. הבה נשתמש בדרך השנייה. אם פותחים את המטמון מהסייר (C:\Winnt\Assembly או C:\Windows\Assembly), אמורים לראות את המכלל Module3Client מופיע בפירוט לצד המכללים האחרים. לחץ לחיצה ימנית עם העכבר על המכלל ובחר ב-Properties. תוכל לראות מידע כגון ערך מפתח ציבורי, מספר גירסה ומיקומו הפיסי של המכלל בדיסק הקשיח שלך. המפתח הציבורי יהיה שונה ממחשב למחשב, אך הנקודה העיקרית היא שמפתח זה יהיה זהה לזה המוצג כאשר מבצעים את היישום Module3ClientApp.

# Versioning Assemblies

מצהר של מכלל מכיל מספר גירסה וגם רשימה של כל המכללים החיצוניים שיש אליהם ייחוס ביחד עם מידע הגרסאות המשויך אליהם. מספרי גרסאות מחולקים לארבעה מקטעים ומקבלים את הצורה הבאה:

<major><minor><build><revision>

כך זה פועל בזמן ריצה: NET משתמשת במידע זה של מספר גירסה בכדי להחליט באיזו גירסה של מכלל מסוים יש להשתמש עם יישום. כפי שתראה בקרוב, התנהגות המחדל - הנקראת מדיניות הבחנת גרסאות - היא שמרגע שיישום מותקן, NET תשתמש אוטומטית בגירסה המאוחרת ביותר של המכללים אליהם פונה היישום, אם הגרסאות מתאימות ברמה של מקטעי major ו-minor. ניתן לשנות התנהגות מחדל זו באמצעות קבצי תצורה.

הבחנת גרסאות קשורה רק למכללים משותפים - למכללים פרטיים אין בכך צורך - והיא אולי הגורם החשוב ביותר היחידי בהחלטה ליצור ולשתף מכללים. לפיכך, הבה נתבונן במעט קוד דוגמה בכדי להמחיש איך כל זה עובד ואיך עובדים עם הבחנת גרסאות של מכלל.

הדוגמה שבה אני עומד להשתמש כאן היא גירסה פשוטה יותר של הדוגמה המצורפת ל-SDK של NET, ואינה כוללת את החומר הנוגע ל- Windows Forms מכיון שברצוני להתמקד בהבחנת גרסאות ובמה שעושה סביבת זמן הריצה בכדי לכפות אותה.

יש לנו שני קבצי ריצה המייצגים שתי חבילות חשבונאות הנקראות Personal ו-business. שני יישומים אלה משתמשים במכלל משותף בשם Account. הפונקציה היחידה של המחלקה Account היא היכולת להודיע על הגירסה שלה כך שאנו יכולים להיות בטוחים שהיישומים שלנו אכן משתמשים בגירסה המיועדת של המחלקה Account. הדוגמה תכיל מספר גרסאות של המחלקה Account כך שתוכל להיווכח איך הבחנת גרסאות פועלת עם מדיניות ברירת המחדל של גירסה, ואיך משתמשים ב-XML ליצירת סמיכות בין יישום וגירסה מסוימת של מכלל.

תחילה ניצור תיקיה בשם Accounting. בתיקיה זו ניצור צמד מפתחות שישמש את כל גרסאות המחלקה Account. לשם כך, נקליד את הפקודה הבאה בשורת הפקודה בתיקיה Accounting:

```
sn /k account.key
```

לאחר שצמד המפתחות נוצר, ניצור תת-תיקיה בשם Personal בתוך התיקיה Accounting. בתוך התיקיה personal ניצור קובץ בשם Personal.cs הנראה כך:

```
// Accounting\Personal\Personal.cs
using System;

class PersonalAccounting
{
   public static void Main()
   {
      Console.WriteLine
         ("PersonalAccounting calling Account.PrintVersion");
      Account.PrintVersion();
   }
}
```

בתוך אותה תיקיית Personal, ניצור תת-תיקיה חדשה בשם Account1000. תיקיה זו תשכן את הגירסה הראשונה של המחלקה Account. לאחר מכן ניצור את הקובץ הבא (Account.cs) בתיקיה Account1000:

```
// Accounting\Personal\Account1000\Account.cs
using System;
using System.Reflection;

[assembly:AssemblyKeyFile("..\\..\\Account.key")]
[assembly:AssemblyVersion("1.0.0.0")]
public class Account
{
   public static void PrintVersion()
   {
      Console.WriteLine
         ("This is version 1.0.0.0 of the Account class");
   }
}
```

המאפיין AssemblyeKeyFile מכוון את מהדר C# לצמד המפתחות שנוצר מקודם והמאפיין AssemblyVersion משמש להצהיר מפורשות על גרסת המחלקה Account. כעת נבנה את הקובץ Account.dll באופן הבא:

```
csc /t:library account.cs
```

מרגע שהמחלקה Account נוצרה, צריך להוסיפה למטמון המכלל הגלובלי:

```
gacutil -i Account.dll
```

אם רוצים, אפשר לאמת שהמכלל Account אכן נמצא במטמון המכלל. כעת ניגש לתיקיה Personal ונבנה את היישום כך:

```
csc Personal.cs /r:Account1000\Account.dll
```

לסיום, נבצע את היישום ונקבל את הפלט הבא:

```
PersonalAccounting calling Account.PrintVersion
This is version 1.0.0.0 of the Account class
```

עד כאן לא חידשנו דבר. עם זאת, הבה נראה מה קורה כאשר מתקינים יישום המשתמש בגירסה חדשה יותר של המחלקה Account.

ניצור תיקיה חדשה בשם Business בתוך התיקיה Accounting. בתוך התיקיה Business ניצור תיקיה בשם Account1001 שתייצג גירסה חדשה של המחלקה Account. מחלקה זו תשכון בקובץ הנקרא Account.cs, והיא תיראה כמעט זהה לגירסה הקודמת:

```
// Accounting\Business\Account1001\Account.cs
using System;
using System.Reflection;

[assembly:AssemblyKeyFile("..\\..\\Account.key")]
[assembly:AssemblyVersion("1.0.0.1")]
public class Account
{
   public static void PrintVersion()
   {
      Console.WriteLine
         ("This is version 1.0.0.1 of the Account class");
   }
}
```

כמו מקודם, נבנה גירסה זו של המחלקה Account באמצעות הפקודות הבאות:

```
csc /t:library Account.cs
gacutil -i Account.dll
```

בנקודה זו אנו אמורים לראות שתי גרסאות של המחלקה Account במטמון המכלל הגלובלי. כעת ניצור את הקובץ Business.cs (בתיקיה Accounting\Business) כדלקמן:

```
// Accounting\Business\Business.cs
using System;

class BusinessAccounting
{
```

```
    public static void Main()
    {
        Console.WriteLine
            ("BusinessAccounting calling Account.PrintVersion");
        Account.PrintVersion();
    }
}
```

נבנה את היישום Business באמצעות הפקודה הבאה:

```
csc business.cs /r:Account1001\Account.dll
```

ביצוע היישום מפיק את הפלט הבא, המאשר את העובדה שהיישום Business משתמש בגירסה 1.0.0.1 של המכלל Account:

```
BusinessAccounting calling Account.PrintVersion
This is version 1.0.0.1 of the Account class
```

כעת נריץ שוב את היישום Personal ונראה מה קרה:

```
PersonalAccounting calling Account.PrintVersion
This is version 1.0.0.1 of the Account class
```

יישומי Personal ו-Business משתמשים גם יחד בגירסה האחרונה של המכלל Account! הסיבה לכך נובעת ממה שנקרא עדכון Quick Fix Engineering (QFE) וממדיניות ברירת מחדל הגירסה.

## QFEs and the Default Version Policy

עדכוני Quick Fix Engineering, או hot fixes, הם תיקונים לא מתוזמנים הנשלחים בכדי לטפל בבעיה עיקרית. מאחר ותיקון כזה אינו משנה את ממשק הקוד, הסיכויים שקוד לקוח יושפע לרעה הם מזעריים. לפיכך, מדיניות מחדל הגירסה היא לשייך באופן אוטומטי את כל קוד הלקוח לגירסה "המתוקנת" החדשה של הקוד אלא אם כן קיים קובץ תצורה עבור יישום, אשר משייך את היישום באופן מפורש לגירסת מכלל מסוימת. גירסה חדשה של מכלל נחשבת QFE (כלומר המכלל תואם) אם המקטע היחיד שהשתנה במספר הגירסה הוא המקטע revision.

## Safe Mode Configuration File

מדיניות המחדל של גירסה עשויה להיות ראויה רוב הזמן, אך קורה שצריכים להגדיר את יישום Personal כך שיפעל רק עם הגירסה שנשלחה בצמוד אליו. כאן נכנסים לתמונה קבצי התצורה XML. קבצים אלה הם בעלי אותו שם כמו של היישום ושוכנים באותה תיקיה. כאשר מתבצע היישום, NET קוראת את קובץ התצורה ואז משתמשת בתגיות XML שבקובץ בכדי להכתיב באיזו גירסה של מכלל להשתמש.

בכדי לציין שיישום ישתמש תמיד בגרסת מכלל שנשלחה אתו, מגדירים "בטוח" (safe mode) את מצב הכריכה של היישום. בלשון המדוברת מתייחסים לזה לעיתים כאל "להכניס את היישום למצב בטוח". בכדי לבחון זאת, ניצור קובץ בשם PersonalAccounting.cfg בתיקיה Accounting/Personal ונקליד לתוכו את הקוד הבא. שים לב לתגית <AppBindingMode>.

```
<?xml version ="1.0"?>
<Configuration>
<BindingMode>
<AppBindingMode Mode="safe"/>
</BindingMode>
</Configuration>
```

כעת אם מבצעים שוב את יישום Personal , ומקבלים את הפלט הבא:

```
PersonalAccounting calling Account.PrintVersion
This is version 1.0.0.0 of the Account class
```

## Specific Versions of Assemblies

כעת נעסוק בסוגיה אחרת הקשורה להבחנת גרסאות. בחלק האחרון של דוגמה זו נשלב שגיאה. בתוך התיקיה Account\Business ניצור תיקיה בשם Account1002. בתיקיה זו ניצור את מחלקת Account הבאה. שים לב שכעת גרסת AccountPrint תוכנתה במכוון כך שתזרוק חריג בכדי לדמות עדכון לא תקין למחלקה Account.

```
// Accounting\Business\Account1002\Account.cs
using System;
using System.Reflection;

[assembly:AssemblyKeyFile("..\\..\\Account.key")]
[assembly:AssemblyVersion("1.0.0.2")]
public class Account
{
   public static void PrintVersion()
   {
      // This is purposely thrown to simulate code failure.
      throw new Exception();

      Console.WriteLine
         ("This is version 1.0.0.2 of the Account class");
   }
}
```

נבנה גירסה זו של מכלל Account כדלקמן:

```
csc /t:library Account.cs
gacutil -i Account.dll
```

אם נבצע כל אחד מהיישומים - Personal או Business - נקבל uncought exception אשר גורמת ליציאה מן היישום. כאן אני מדמה תופעה שכיחה ביותר בהטמעת תוכנה, שבה התקנת משהו שאינו עובד גורמת בסופו של דבר לקריסה של מספר יישומים אחרים. במקרה של יישום Personal, איננו רוצים לחזור למצב בטוח מאחר וברצוננו לבצע את הגירסה המאוחרת ביותר התקינה של Account (גירסה 1.0.0.1) ומצב בטוח גורם לנו לעבוד עם הגירסה הראשונה (1.0.0.0).

ושוב, התשובה טמונה בקובץ התצורה של היישום.

כעת נשנה את Accounting\PersonalAccounting.cfg כך שתגיות XML שלו יראו כך:

```
<?xml version ="1.0"?>
<Configuration>
<BindingPolicy>
<BindingRedir Name="Account"
            Originator="32ab35a4550339b1"
            Version="*"
            VersionNew="1.0.0.1"
            UseLatestBuildRevision="no"/>
</BindingPolicy>
</Configuration>
```

שים לב כי המפתח שהוגדר עבור התכונה Originator הוא זה שנוצר במערכת שלי. עליך להחליף אותו בזה שהתקבל כאשר יצרת את הקובץ Account.key. תוכל לאתר את הערך הנדרש על ידי פתיחת מטמון המכלל ואיתור המחלקה Account.

לסיום, נבצע את יישום Personal שוב, ונראה שהוא פועל ללא תקלות וכן שהיישום אכן משתמש במחלקת Account מהגירסה המבוקשת (1.0.0.1).

# סיכום

בפרק זה עסקתי ביתרונות השימוש העיקריים ב-Assemblies, כולל אריזה, הטמעה והבחנת גרסאות. כמו כן עסקתי במכללי קובץ אחד ומכללי קבצים מרובים ובכלי Assembly Generation (al.exe), איך ליצור ולשתף מכללים באמצעות הכלי Strong Name (sn.exe), ובכלי Global Assembly Cache (gacutil.exe). הדגמתי איך שולטים במדיניות הבחנת הגרסאות של ברירת המחדל בה משתמשת NET באמצעות יצירת קבצי תצורה מבוססי XML.

# Index

*Note: Italicized page references indicate figures, tables, or code listings.*

## Special Characters


+ (addition operator), 209, 218–219, 229, 289, 291
& (address of operator), 372
& (and operator), 291
= (assignment operator), 209, 210, 221–225, 229–232, 291
^ (caret operator), 291
: (colon), 163, 180
(,) (comma)
  in arrays, 152
  as operator in for statements, 250–251, 291
{} (curly brackets or braces operator), 61, 89, 236, 339
* (dereference operator), 372
-> (dereferencing and member access operator), 372
/ (division operator), 209, 218–219, 291
. (dot operator), 213
== (equal to operator), 210, 238, 291
> (greater than operator), 227, 291
>= (greater than or equal to operator), 227, 291
++ (increment operator), 214, 225–226, 291
< (less than operator), 227, 291
<= (less than or equal to operator), 227, 291
% (modulus operator), 209, 218–219
* (multiplication operator), 209, 218–219, 291
!= (not equal to operator), 210, 291
! (not operator), 291

() (parenthesis operators), 213, 221, 292
; (semicolon), 236
[] (square brackets operator), 164, 213, 292
+ (string concatenation operator), 210, 229
+= (string operator), 210
- (subtraction operator), 209, 218-219, 291
- (unary minus operator), 209, 219-221, 291
+ (unary plus operator), 209, 219–221, 291

## A

Abort method, 328, 330
AboveNormal thread priority, 330
abstract classes, 183, 184
abstractions
  benefits of, 28, 33
  designing, 31–33
  operator overloading as, 288
  properties as, 149
abstract keyword, 148
access modifiers
  assemblies and, 393, 394
  class members and, 28, 89, 91–92, 91
  encapsulation and, 31
  inheritance and, 34
  interface methods and, 184
  querying types for, 349
  type safety and, 88
accessor methods, 143–145
Activator class, 357–359, 383
addition operator (+), 209, 218–219, 229, 289, 291
address of operator (&), 372
aggregation, 97, 119, 203–204, 288
al.exe (Assembly Generation tool), 390, 394–395
aliases, 63, 81, 81–82
  GetType method and, 349
  using directive and, 86
ambiguity, class, 64
ambiguity, name, 196–200
and operator (&), 291
annotations. See attributes
ANSI character set, 368–369
apartments, COM threading, 385–386

ApartmentState property, 386
<AppBindingMode> tag, 402–403
AppDomain class, 326–327, 361
applications
  code (see code)
  command-line arguments, 93–94
  deployment, 391
  development (see C# programming)
  name of currently executing, 354
  running, 59–60
arguments. See also parameters
  application command-line, Main method and, 93–94
  delegate, 316
  displaying, for methods, 57, 69
  Main method, 61
  naming conventions, 73
  overloaded operator, 288
  parenthesis operator and, 213
  user-defined conversions, 293
Array class, 150, 152, 154–155
ArrayList class, 158–160
arrays, 150–156
  of command-line arguments, 93–94
  declaring, 150–151
  indexers as smart, 90, 157 (see also indexers)
  iteration of, 252–254
  jagged (arrays of arrays), 155–156
  multidimensional, 152–154
  of objects, 30–31, 40
  objects as, 156–160
  of parameters, 131–132
  querying for rank of, 154–155
  as reference types, 79
  single-dimensional example, 151–152
  System.Array as base class, 150
as operator, 84, 168, 190–193, 201

assemblies, 352–356, 389–404
benefits of, 391–392
building, 392–395
deployment, 52, 391–392
global assembly cache and, 397–399
install-time code generation and, 49
iterating through types of, 352–355
listing modules of, 355–356
manifest data, 390–391
multifile, 393–395
overview of, 389–391
packaging, 391
Quick Fix Engineering updates (QFEs), 402
reflection API and, 347
safe mode configuration files, 402–403
shared, 395–397
single-file, 392–393
using specific versions of, 403–404
versioning, 392, 399–404
AssemblyBuilderAccess.Run value, 361
AssemblyBuilder class, 352
Assembly Cache Viewer (shfusion.dll), 397, 398
Assembly class, 352, 397
Assembly Generation tool (al.exe), 390, 394–395
AssemblyKeyFile attribute, 395–396, 400
AssemblyName class, 361
AssemblyVersion attribute, 400
assignment operator (=), 209, 291
associativity of, 212
compound, 221–225
simple, 229–232
associativity, left and right, 211–212
asynchronous event handling, 90, 315–318
asynchronous processing, 302, 344
Attribute class, 165
attributes, 163–180
assembly manifest and custom, 391
benefits of, 163–165
class, 166–168
defining, 165–166
defining usage of, with AttributeUsage attribute, 176–179

exported DLL functions and, 366–369
field, 171–172
identifiers, 179–180
method, 169–170
parameters, 173–175
querying about, 166–172
querying types for, 349–351
AttributeTargets enumeration, 176–177
AttributeUsage attribute, 176–179
defining attribute targets, 176–177
single-use and multiuse attributes, 178–179
specifying attribute inheritance, 179

## B

background threads, 344
base classes
constructors, 99
derived classes and, 34
exception handling and, 281
polymorphism and, 40
substitutability and, 37, 83
virtual methods and (see virtual methods)
base class library (BCL), 44
BelowNormal thread priority, 330
benchmarking, threads and, 345
binary operators, 211–212
overloading, 288–289, 291
binding
early, 136
early, to COM components, 380–381
late, to COM components, 383–385
polymorphism and late, 137–140
RCWs and, 378
reflection and late, 357–359
safe mode, 402–403
Boolean expressions and values
do/while statement and, 247
if statement and, 235, 238
methods and, 73
relational operators and, 226
while statement and, 245
boxing, 79–80

braces operator ({}), 61, 89, 236, 339
branching. See jump statements
break statements, 254–256
  breaking out of infinite loops, 255–256
  switch statement and, 240, 244–245
Building Block services, .NET, 43
build numbers, 390, 399

## C

C/C++ programming
  code snippets, 26
  constructor initializers, 102
  cout function, 46
  function pointers, 301
  garbage collection, 114
  if statement, 238
  inline programming, 60
  instantiation, 96
  interfaces as abstract classes, 183
  object-oriented programming and, 23
  pointer operators, 372
C++ FAQs (Cline and Lomow), 37
C# programming
  arrays, 150–156
  assemblies (see assemblies)
  attributes (see attributes)
  C/C++ and (see C/C++ programming)
  choosing editors, 55–58
  class ambiguity, 64
  classes and members, 61 (see also classes; class members)
  code snippets, 26 (see also code)
  code walk-through, 60–64
  command-line compiler, 58–59 (see also compiling)
  defining namespaces, 70
  delegates and event handlers (see delegates)
  enforcement of if rules, 238–240
  environment (see Microsoft .NET Framework)
  errors, 65–66, 66 (see also errors; exception handling)
  flow control (see flow control statements)
  guidelines, 70–74
  "Hello, World" application, 58
  IL Disassembler (ILDASM), 66–69 (see also ILDASM (IL Disassembler))

indexers, 156–160 (see also indexers)
inline, as one-stop, 60–61
interfaces (see interfaces)
interoperating with unmanaged code (see interoperability; unmanaged code)
Main methods, 61
methods (see methods)
multithreaded (see threading)
namespaces and using directive, 62–63, 64
naming conventions, 70–74 (see also naming conventions)
non-object-oriented vs. object-oriented, 25–28 (see also object-oriented programming)
operator overloading, 287–292 (see also operator overloading)
operator precedence, 211, 211
operators and expressions (see operators)
pointer operators, 372
properties, 143–150 (see also properties)
querying metadata with reflection (see reflection)
running applications, 59–60
skeleton code for, 63
System.Console.WriteLine method, 62
type system (see Common Type System (CTS), .NET; types)
user-defined conversions, 292–299
writing simple applications, 55–60
cache, global assembly, 397–399
cache manager, assembly, 361
callback functions, delegates and, 369–270
callback methods, delegates as, 301–305
call stack, exceptions and, 279–280, 285
callvirt MSIL opcode, 139
camel casing, 72
caret operator (^), 291
Caron, Rob, 72
case statements, 240. See also switch statement
combining, 242–244
casing, Pascal and camel, 72
caspol.exe (Code Access Security Policy) tool, 354
castclass MSIL opcode, 190
casting. See type casting

catch blocks
  design issues, 285–286
  multiple exception types, 280–281
  running code after, 269–270
catch keyword, 267–268
character sets, 368–369, 371
CharSet parameter, 368, 371
checked operator, 218
circular reference problem, 106
classes, 89–121
  abstract, 183, 184
  access modifiers, 28, 91–92, 91 (see also access modifiers)
  aliases, 63
  attributes, 166–168 (see also attributes)
  class ambiguity and namespaces, 64
  constants vs. read-only fields, 102–105
  constructors, 95–102
  defining, 89
  defining members and, 61
  deployment, 391
  encapsulation and, 31–33
  exception (see Exception class)
  inheritance and, 34–37, 117–121
  interfaces vs., 73, 181–182, 183 (see also interfaces)
  Main methods, 92–95
  marking, for deterministic finalization, 115–116
  members, 90–91 (see also class members)
  methods (see methods)
  naming conventions, 73
  .NET Framework libraries, 46–47
  objects as instantiation of, 29–31 (see also objects)
  objects vs., 29
  polymorphism and, 37–40
  properties (see properties)
  as reference types, 79
  sealed, 120–121
  self-describing, 52, 163–164, 391
  thread safety and .NET, 343
  as types, 29, 77 (see also types)
  user-defined conversions, 292–299
class keyword, 89, 163

class libraries, .NET Framework, 44, 46–47
class members
  defining, 61
  delegates as static, 305–107
  instance members and constructor initializers, 102
  naming conventions, 74
  public, 30, 31, 33
  static members vs. instance members, 97–99
  static methods and access to, 142
  type system and, 90–91
cleanup, object. See resource management
client programmer issues, 32–33, 47, 299
Close method, 343
CLR (Common Language Runtime), .NET, 44, 45, 77, 265, 285, 286
CLS (Common Language Specification), .NET, 45, 86–87, 265
Coad/Yourdon definition of problem domain, 25
code. See also applications
  cache, 397–399
  calling new, from old, 37, 40
  class ambiguity, 64
  compiling (see compiling)
  creating and executing, at run time, 360–363
  defining classes and members, 61
  injecting custom, into class code path, 302
  inline programming and one-stop programming, 60–61
  install-time generation of, 49
  legacy, 365, 386–387 (see also interoperability)
  Main method, 61
  maintenance, 273, 274
  managed (see managed code)
  metadata (see metadata; reflection)
  namespaces and using directive, 62–63
  readability and exception handling, 274–275
  reusability, 33, 35, 36, 375
  running, after catching exceptions, 269–270
  skeleton, for application development, 63
  System.Console.WriteLine method, 62
  unmanaged, 52–53, 365 (see also unmanaged code)
  unsafe (see unsafe code)
  walk-through, 60–64
Code Access Security Policy tool (caspol.exe), 354

code pitching, 49
CodeWright, 58
cohesion, tight, 273
collection, garbage. See resource management
collections, 25
  iteration of, 252–254
colon (:), 163, 180
combining. See composition
COM components. See COM interoperability
COMException class, 381
COM interoperability, 375–376
  COM Interop layer and, 375
  COM threading models and, 385–386
  early binding to COM components, 380–381
  example COM component and, 376–377
  generating metadata from COM type libraries (typelibs), 377–379, 378, 379
  late binding to COM components, 383–385
  managed code, unmanaged code, and, 53
  using dynamic type discovery to select COM interfaces, 382–383
COM Interop layer, .NET, 375. See also COM interoperability
comma (,)
  in arrays, 152
  as operator in for statements, 250–251, 291
command-line arguments
  Global Assembly Cache tool, 397–398
  Main methods and application, 93–94
Common Language Runtime (CLR), .NET, 44, 45, 77, 265, 285, 286
Common Language Specification (CLS), .NET, 45, 86–87, 265
Common Type System (CTS), .NET, 77–88. See also types
  benefits of, 86–88
  boxing and unboxing, 79–80
  casting between types, 82–84
  class member types, 90–91
  everything as objects, 77–78
  language interoperability and, 86–87
  namespaces, 84–86
  reference types, 79
  singly rooted object hierarchy of, 87
  System.Object class as root type, 80–81, 80–81, 87
  System.Type class (see Type class)

type safety and, 87–88
types and aliases, 81–82, 81–82
as unified type system, 50
value types, 78
company name
in assembly manifest, 391
as namespace name, 70, 72, 85
Compare method, 238
comparison method, 80
comparison operators (==, !=), 227–229
compatibility, 77–78, 87, 106–107
compiling
assemblies, 389–390, 392–395, 400
attribute identifiers and, 179–180
C# command-line compiler, 58–59, 59
errors, 65–66, 66
JIT (just-in-time), 48–49, 397
Microsoft intermediate language (MSIL) and, 48–49
modules, 356
multiple Main methods, 95
.NET Framework class libraries, 47
pinned variables, 375
properties and, 146–147
unsafe code, 373
components, COM. See COM interoperability
components, .NET. See assemblies
composition
of case labels, 242–244
of delegates, 309–315
of interfaces, 203–204
resource management and, 115
compound assignment operators, 221–225, 291
compound operators, overloading, 289
compound statements, 236
concatenation operator, string (+), 210, 229
concurrency, threads and, 344
conditional operators, 212
conditional statements. See flow control statements
configuration files, safe mode, 402–404
console applications, 59
Console class, 60, 62

constants
  as class members, 90
  read-only fields vs., 102–105
const keyword, 90, 102–103
constructors, 95–102
  attribute, 173–175
  Exception object, 276–279, 282
  inheritance and, 118
  initializers, 99–102
  instantiating, 96–97
  mutex, 341
  naming, 95–96
  overloading, 129–130
  read-only fields and, 103
  static members and instance members, 97–99
  throwing exceptions from, 276
containment, 97
context, exception, 266
  handling errors in correct, 273–274
CONTEXT structure, 325
context switching, 107, 324–325
continue statement, 257–258
controlling program flow. See flow control statements
conversions between types, 79–80, 191, 292–299
cooperative multitasking, 324
copying objects, 81
_CorExeMain function, 48
costs. See also performance
  code maintenance and, 273, 274
  threads and, 345
cout function, C++, 46
CPU utilization, threads and, 344
CreateInstance method, 383
CreateType method, 362
critical sections, 335. See also thread safety
cross-language interoperability, 45. See also interoperability
csc.exe compiler, 58–59, 59, 356. See also compiling
cs extension, 56
CTS. See Common Type System (CTS), .NET
curly brackets or braces operator ({}), 61, 89, 236, 339
Current.Thread method, 327

custom attributes, 391
custom exception classes, 281–283
cycles, reference counting and, 108

## D

data
  fields as, 90
  metadata as embedded, 50 (see also reflection)
  serialized, 277
database connections, resource management and, 107
data structures. See structures
declaration
  arrays, 150–151
  exported DLL functions, 366–369
  instantiation and, 30, 97
  interfaces, 183–184
deep copies, 81
default switch statements, 241
default values, 173–174
default versioning policy, 399, 402
DefineDynamicAssembly method, 361
DefineDynamicModule method, 361
DefineMethod method, 362
DefineType method, 361
definition
  attribute targets, 176–177
  classes and members, 61, 89–91, 163
  events, 315–318
  indexers, 157–158
  namespaces, 70
delegate keyword, 303
delegates, 301–319
  as callback methods, 301–305, 369–370
  as reference types, 79
  composing, 309–315
  creating, only when needed, 307–309
  defining, as static members, 305–307
  defining events with, 315–318
threading and, 326
deployment, assembly, 52, 352, 390, 391–392
dereference operator (*), 372

dereferencing and member access operator (->), 372
derived classes. See also inheritance
  base classes vs., 34
  Exception classes, 281–283
  signifying, with colon, 163
  substitutability and, 37, 83
  virtual methods and (see virtual methods)
design guidelines
  abstractions, 31–33
  catch blocks, 285–286
  exception handling, 283–286
  indexers, 160
  operator overloading, 292
  resource management, 116–117
  threading, 343–345
  try blocks, 283–285
destroying threads, 328–330
destructors, 105
deterministic finalization
  combining tracing and reference counting collection, 114
  design pattern, 116–117
  marking classes for, 115–116
  performance and, 108–114
  reference counting collection, 108
  resource management and, 107–108 (see also resource management)
  tracing collection, 108
development. See C# programming; Microsoft .NET Framework
device software, .NET, 43
Dijkstra, Edsger, 258–259
dimensions, querying for array, 154–155
Directory class, 359
disassembler. See ILDASM (IL Disassembler)
Dispose design pattern, 116–117
Dispose method, 117
division operator (/), 209, 218–219, 291
DllImport attribute, 366–371
DLLs (dynamic-link libraries)
  assemblies and, 389–390
  assembly versioning and, 392
  calling unmanaged functions of, 53
  creating, 392–393
  declaring exported functions, 366–369
  managed vs. unmanaged, 69, 366

do/while statement, 247–249
dot operator (.), 213
downcasting, 82–83. See also type casting
dup MSIL opcode, 224
dynamic boost values, 334
Dynamic Help, 57
dynamic type discovery, 382–383

## E

early binding, 136
  to COM components, 380–381, 383
EconoJIT compiler, 49
editors, 55–58
  Notepad, 55–56
  third-party, 58
  this book and, 58
  Visual Studio 26, 56–57
  Visual Studio.NET, 57
efficiency, 24
else clauses, multiple, 237–238. See also if statement
Emit method, 362
encapsulation
  benefits of abstractions, 33
  designing abstractions, 31–33
  inline programming and, 60–61
  object-oriented programming and, 26
  operator overloading and, 292
Enter method, 337
Enterprise Servers, .NET, 44
.entrypoint MSIL keyword, 69
EntryPoint named parameter, 368
Enum functions, 369–370
EnumWindows function, 370
Equals method, 80, 228–229
equal to operator (==), 210, 238, 291
error codes, returning, 270–271
  exception handling vs., 271–273
error-handling techniques, 270–276
  code readability and, 274–275
  context and, 273–274
  exception handling vs., 271–273 (see also exception handling)
  return codes and, 270–271
  throwing exceptions from constructors, 276

errors
  COM component, 379, 381
  compile-time, 65–66, 66
  named-parameter, 175
  run-time exceptions, 83–84 (see also error-handling techniques; exception handling)
EventArgs class, 316
event keyword, 318
events
  as class members, 90
  defining, with delegates, 315–318 (see also .delegates)
  exceptions vs., 265–266
Exception class, 267, 276–283
  catching multiple exception types, 280–281
  constructing exception objects, 276–279
  deriving custom exception classes, 281–283
  handling errors in correct context and, 274
  naming custom exception classes, 282
  using StackTrace property, 279–280
exception handling, 265–286
  array out-of-bounds errors, 160
  benefits of, over return codes, 271–273
  catching exceptions, 267–268
  cleaning up with finally, 269–270
  code readability and, 274–275
  COM components and, 379, 381
  comparing techniques, 270–276
  context and, 273–274
  design guidelines, 283–286
  nonworking updates and, 403–404
  overview of, 265–266
  resource management and, 109
  rethrowing exceptions, 268–269
  return statement, 263
  syntax, 267–270
  throwing exceptions, 267
  throwing exceptions from constructors, 276
  type casting and, 83–84
  using System.Exception class (see Exception class)
executables, Win32-portable (PEs), 48, 347, 352, 389. See also assemblies
executing code at run time, 360–363

EXEs
  assemblies and, 389
  compiling (see compiling)
  managed vs. unmanaged determination of, 69
  metadata in, 50 (see also reflection)
Exit method, 337
exit statement, 259
explicit conversions, 293
explicit interface member name qualification, 193–200
  name ambiguity, 196–200
  name hiding, 193–196
exported DLL functions, 366–371
expressions, methods in Boolean, 73. See also operators
extensibility, 24, 36, 50, 143, 274
extern access modifier, 366
external assemblies, 390

## F

fall-through and switch statements, 244–245
false value, 226, 237
FieldInfo class, 172
fields
  attributes, 171–172
  as class members, 90
  constants vs. read-only, 90, 102–105
  properties as smart, 90, 149–150 (see also properties)
files
  assemblies (see assemblies)
  key, 395–397
  resource management and, 107
  safe mode configuration, 402–404
finalization. See deterministic finalization
Finalize method, 81, 108, 116
finally keyword, 263, 267, 269–270, 328
first-class concept, 183
fixed keyword, 372, 373–375
fixes, QFE, 402
flow control statements, 235–264
  iteration statements, 245–254
  jump statements, 254–263
  selection statements, 235–245

folders
  assembly deployment and, 391
  global assembly cache and, 397–399
foreach statement, 252–254
for statement, 249–251
  empty, 256
  nested loops, 250
  using comma operator, 250–251
Framework. See Microsoft .NET Framework
function pointers, C++, 301
functions. See also methods
  calling unmanaged, 53
  declaring exported DLL, 366–369
  marshalling parameters for exported DLL, 370–371
  using callback, 369–370

## G

gacutil.exe (Global Assembly Cache utility), 397–399, 400
Garbage Collector (GC), .NET, 105. See also resource management
  pinning and, 372, 373–375
GetCustomAttributes method, 168, 170
GetFiles method, 359
get method (getter), 90, 144–145
GetHashCode method, 80
GetILGenerator method, 362
GetLength method, 154
GetMembers method, 215
GetMethod method, 359
GetMethods method, 170, 215
GetModules method, 356
GetProcessName method, 354
GetTypeFromProgID method, 383–384
GetType method, 80, 170, 215, 348, 349, 362
GetTypes method, 354
GetWindowText function, 370
global assembly cache, 397–399, 400
goto statement, 258–263
  history of, 258–259
  using, 259–263
greater than operator (>), 227, 291
greater than or equal to operator (>=), 227, 291
guidelines, design. See design guidelines

guidelines, naming. See naming conventions
guidelines, programming, 70–74
  defining namespaces, 70
  naming conventions, 70–74

## H

handles, resource management and, 107
Harry, Brian, 106
hash, signed assembly, 390
hash codes, 80
"Hello World" application, 58
  code walk-through, 60–64
  in ILDASM, 67–69, 67, 68
Help, Dynamic, 57
help, online, 65–66, 66
hiding information. See encapsulation
hiding names with interfaces, 193–196
hierarchies, class, 34
Highest thread priority, 330
highlighting, syntax, 56, 57
history of goto statement, 258–259
history of resource management, 106–107
hot fixes, 402
HRESULT type, 179–180
Hungarian notation, 71–72

## I

icons, ILDASM, 68
identifiers, attribute, 179–180
if statement, 235–240
  C# enforcement of rules, 238–240
  multiple else clauses, 237–238
ILDASM (IL Disassembler), 66–69
  COM type library metadata and, 379, 379
  "Hello, World" application and, 67–69, 67, 68
  metadata and, 51, 51
  using, to determine whether EXE or DLL is .managed, 69
ILGenerator class, 362
implementation, interface, 182, 184–193
  querying by using as, 190–193
  querying by using is, 186–190

implicit conversions, 293
increment operator (++), 214, 225–226, 291
index argument, 157
indexers, 156–160
  as class members, 90
  defining, 157–158
  design guidelines, 160
  example, 158–160
  square brackets operator and, 213
infinite loops, breaking out of, 255–56
information association. See attributes
information hiding. See encapsulation
infrastructure, .NET, 44. See also Microsoft .NET Framework
inheritance, 117–121
  attribute, 179
  defining proper, 37
  interfaces and, 181, 182, 200–203
  method overriding and, 118
  multiple interfaces and, 119–120
  object-oriented programming and, 34–37
  of properties, 148
  rules of substitutability and, 117–118
  sealed classes and, 120–121
initialization
  constructors and, 95, 99–102
  for statement, 249
  lazy, 149–150
  method parameters and, 125–126, 127
  static members, 99
inline programming, 60–61
InnerException property, 278
inner exceptions, 277–278
installing assemblies, 397–398
install-time code generation, 49
instance members
  constructor initializers and, 102
  static members vs., 97–99
  static methods and, 142
InstanceOf method, 382

instances
objects as, 29
retrieving types of, 348
instantiation
array, 150–151
of classes, objects as, 29–31
constructor, 96–97
thread objects, 327
Int32 type, 78
IntelliSense, 50, 57, 73
interfaces, 181–205
as abstract classes, 183
abstraction and, 32
classes vs., 73, 181–182
COM, dynamic type discovery and, 382–383
combining, 203–204
as contracts, 37, 181
declaring, 183–184
deterministic finalization and, 116
encapsulation and, 31
example, 182–183
explicit member name qualification, 193–200
implementing, 184–193
implementing vs. inheriting from, 182
inheritance and, 200–203
multiple, 119–120
naming conventions, 73
object-oriented programming and, 25–26
operator overloading, user-defined conversions, and, 287, 299
as reference types, 79
internal access modifier, 88, 91, 393, 394
Internet devices, .NET, 43
interoperability
COM (see COM interoperability)
Common Language Runtime and, 45
Framework class libraries and, 46
legacy code and, 386–387
type system and, 86–87
unmanaged code and, 52–53, 365 (see also unmanaged code)
interrogating types, 349–351

Interrupt method, 328
intranets, security and, 354
int type, 80
InvalidCastException, 84, 382
InvokeMember method, 384
Invoke method, 359
IO.Directory class, 359
isinst MSIL opcode, 190
is operator, 170, 186–190, 198
iteration
  through arrays, 252–254
  through types of assemblies, 352–355
iteration statements, 245–254
  do/while statement, 247–249
  foreach statement, 252–254
  for statement, 249–251
  while statement, 245–247

## J

jagged arrays, 155–156
Java, object-oriented programming and, 23
Java Virtual Machine (JVM), 44
JIT (just-in-time) compilers, 48–49, 397
Join method, 330
jump statements, 254–263
  break statement, 254–256
  continue statement, 257–258
  goto statement, 258–263
  return statement, 263
  switch statement and, 40
just-in-time (JIT) compilers, 48–49, 397

## K

key files, 395–397, 399–402
keywords, 56–57

## L

labels
  combining *case,* 242–244
  *goto* 259–260

languages
benefits of multiple, 47
C/C++ (*see* C/C++ programming)
C# (*see* C# programming)
interoperability (*see* interoperability)
Microsoft intermediate language (MSIL), 48–49
.NET Framework class libraries and, 47
object-oriented programming and, 23–24, 31 (*see also* object-oriented programming)
types and, 77–78
late binding
to COM components, 383–385
polymorphism and, 137–140
with reflection, 357–359
lazy garbage collection, 105
lazy initialization, 149–150
*ldstr* MSIL opcode, 69
left associativity, 211–212
legacy code, 365, 386–387. *See also* interoperability
*Length* property, 152
less than operator (<), 227, 291
less than or equal to (<=), 227, 291
lexical analyzer, 210
libraries
COM type (*see* type libraries (typelibs), COM)
DLLs (*see* DLLs (dynamic-link libraries))
.NET Framework class, 46–47
lifetimes
object, 378
thread, 327–328
*LoadFrom* method, 354, 359
locks, monitor, 337, 339
*lock* statement, 339–341
loops. *See also* iteration statements; jump statements
breaking out of infinite, 255–256
nested *for,* 250
lowercase, 72
*Lowest* thread priority, 330
*lvalue* value, 221, 229, 375

## M

*Main* methods
  arguments and return values for, 61
  classes and, 92–95
  command-line arguments, 93–94
  ILDASM and, 51, *51,* 68–69, *68*
  multiple, 94–95
  return values, 94
  threading and, 326
maintenance costs, 273, 274
major version numbers, 390, 399
managed code
  Common Language Runtime and, 45
  determination of, using ILDASM, 69
  unmanaged vs., 53 (*see also* unmanaged code)
manifests, assembly, 389–391
*MarshalAs* attribute, 371
marshalling, 370–371, 381
mathematical operators, 218–226
  compound assignment, 221–225
  increment and decrement, 225–226
  unary, 219–221
member access and dereferencing operator (->), *372*
members. *See* class members
*MemberwiseClone* method, *81*
memory
  manipulating, with unsafe code, 371–375
  resource management and, 49, 107 (*see also* resource management)
*Message* property, 277
metadata
  generating, from COM type libraries (typelibs), 377–379, *378, 379*
  reflection and, 50–51, *51,* 347 (*see also* reflection)
*MethodBuilder* class, 362
*MethodInfo* class, 170, 359, 362
*.method* MSIL keyword, 69
methods, 123–142
  accessor, 143–144
  attributes, 169–170
  as class members, 90
  constructor (*see* constructors)
  delegates as callback, 301–305

destructor, 105
displaying arguments of, 57, 69 (*see also* arguments)
ILDASM and, 68–69, *68*
just-in-time compiling and, 49
*Main* (*see Main* methods)
marshalling parameters, 371
naming conventions, 72, 73
operator overloading and, 287, 288
overloading, 99, 128–130
overriding, 118, 132–134
*ref* and *out* parameters, 123–128 (*see also* parameters; return values)
static, 140–142
*System.Object, 80–81*
variable parameters, 131–132
variables and, 28
virtual (*see* virtual methods)
MicroEdge Visual SlickEdit, 58
Microsoft intermediate language (MSIL), 48–49
ILDASM and, 66–69, *67, 68*
Microsoft .NET Framework, 43–53. *See also* C# programming
class libraries, 46–47
COM Interop layer, .NET, 375 (*see also* COM interoperability)
Common Language Runtime (CLR), 44, 45, 77, 265, 285, 286
Common Language Specification (CLS), 45, 86–87, 265
deployment, 52
Garbage Collector (GC), 105 (*see also* resource management)
IL Disassembler (ILDASM), 66–69 (*see also* ILDASM (IL Disassembler)
interoperability and unmanaged code, 52–53
metadata and reflection, 50–51
Microsoft intermediate language (MSIL) and JITters, 48–49
Microsoft .NET platform and, 43–44
naming guidelines, 72
Platform Invocation Services, 365–371
reflection API hierarchy, 166, 347–348, *348* (*see also* reflection)
SDK Documentation, 65–66, *66,* 72
security, 51–52
thread safety, 343
unified type system, 50 (*see also* Common Type System (CTS), .NET)
Windows DNA and, 44–45
Microsoft .NET platform, 43–44
Microsoft Visual Basic, 23, 106–107, 112

Microsoft Visual Basic.NET, 112
Microsoft Visual C++, 56, 102
Microsoft Visual Studio 26, 56–57
Microsoft Visual Studio.NET, 44, 50, 57
Microsoft Windows Explorer, 397–399
  multitasking, 324
  Registry (*see* Registry)
  Windows DNA, 44–45
  Windows.NET, 44
minor version numbers, 390, 399
modifiability, 24
*ModuleBuilder* class, 361
modules
  assemblies and, 389–390
  creating, 392–393
  creating assemblies with multiple, 393–395
  listing assembly, 355–356
modulus operator (%), 209, 218–219
*Monitor* class, 335–339
monitor locks, 339–341
MTAs (multithreaded apartments), 385–386
multidimensional arrays, 152–154
multifile assemblies, 392, 393–395
multiple *else* clauses, 237–238. *See also if* statement
multiple exception types, catching, 280–281
multiple interfaces, 119–120
multiple *Main* methods, 94–95
multiple statements in *for* statements, 250–251
multiplication operator (*), 209, 218–219, 291
multitasking and threads, 324
multithreaded apartments (MTAs), 385–386
multithreaded programming. *See* threading
multiuse attributes, 178–179
*Mutex* class, 341–343

## N

name collision, 70, 196–200
named parameters, 173–175
  common mistakes with, 175
  *DllImport* attribute, 367–368, 371

names
ambiguity, 196–200
assembly, 390
conventions (*see* naming conventions)
of currently executing applications, 354
explicit interface member qualification, 193–200
of exported DLL functions, 367–368
hiding, 193–196
retrieving object, *81*
retrieving types from, 349
shared assembly, 390, 395–397
namespaces
class ambiguity and, 64
defining, 70
interoperability, 367, 371, 381
naming conventions, 70, 72–73, 85
*System* namespace, 62–63
*System.Reflection*, 347–348, *348*
*System.Reflection.Emit*, 360–363
*System.Runtime.InteropServices*, 367, 371, 381
*System.Threading*, 325
type system and, 84–86
*using* directive and, 62–63, 85–86
naming conventions. *See also* names
attributes, 166
benefits of, 70–71
class ambiguity and, 64
classes, 61, 73
class members, 61, 74
constructors, 95–96
delegates, 307
derived exception classes, 282
Hungarian notation, 71–72
interfaces, 73
method arguments, 73
methods, 73
namespaces, 70, 72–73, 85
.NET Framework guidelines, 72
Pascal casing and camel casing, 72
standard, 71–74
nested *for* loops, 250

.NET. *See* Microsoft .NET Framework; Microsoft .NET platform
.netmodule extension, 390
*new* operator, 30, 96, 133, 214, 381
non-object-oriented programming, 25–28
*Normal* thread priority, 330
Notepad, 55–56
not equal to operator (!=), 210, 291
not operator, 291
*NullReferenceException,* 84
*null* value, 78, 79, 84, 168, 191, 396
numbers, version, 390, 399, 402

## O

*Object* class, 72, 87, 384
object-oriented programming, 23–41. *See also* C# programming
  clients and, 33
  concepts, 23–31
  encapsulation, 31–33
  inheritance, 34–37
  instantiation, 29–31
  languages and, 23–24
  objects, 25–31
  objects vs. classes, 29
  polymorphism, 37–40
objects
  as arrays, 156–160 (*see also* indexers)
  assignment and, 229–232
  comparing, 227–229
  concept of, 25–31
  example, 26–28
  as instantiation of classes, 29–31 (*see also* classes)
  lifetime management, 378
  object cleanup (*see* resource management)
  pinning, 372, 373–375
  references to, 79
  *System.Object* class as root of all, 80–81, *80–81,* 87
  type system and, 77–78 (*see also* Common Type System (CTS), .NET)
object types, 23, 77–78
one-stop programming, 60–61
opcodes, MSIL, 66, 69
*OpCodes* class, 362
operands, 209

*operator* keyword, 293
operator overloading, 91, 287–292
  design guidelines, 292
  object comparison, 229
  overloadable operators, 291
  restrictions, 291–292
  syntax and example, 288–291
operator precedence, 210–213
  C# and, 211, *211*
  left and right associativity, 211–212
  practical usage, 212–213
operators, 209–233
  *as* operator, 84, 168, 190–193, 201
  as class members, 91
  defined, 209–210
  *is* operator, 170, 186–190, 198
  mathematical, 218–226
  *new* operator, 30, 96, 133, 214, 381
  overloadable, 291
  overloading (*see* operator overloading)
  pointer, *372*
  precedence of, 210–213
  primary expression, 213–218
  relational, 226–229
  simple assignment, 229–232
*Originator* attribute, 404
*out* method parameters, 123–128
overflow checking, 218
overloading methods, 99, 128–130
overloading operators. *See* operator overloading
*override* keyword, 137, 148
overriding methods, 118, 132–134

## P

**S**packaging, assembly, 391
parameters. *See also* arguments; return values
  attribute, 173–175
  common mistakes with named, 175
  *DllImport,* 367–369
  marshalling and, 371
  marshalling DLL function, 370–371

method overloading and, 128–130
positional and named, 173–174
*ref* and *out,* 123–128
valid types, 175
variable number of, 131–132
*params* keyword, 131
parenthesis operators (), 213, 221, 292
Pascal, object-oriented programming and, 24
Pascal casing, 72
passing values. *See* arguments; parameters; return values
Passport.NET, 43
pausing threads, 327
PE files (Win32-portable executables), 48, 347, 352, 389. *See also* assemblies
performance
assemblies and, 391
binding and, 383
COM threading models and, 385
just-in-time compiling and, 49
objects and, 78
resource management and, 108–114
threads and, 345
permissions. *See* security
pinning, 372, 373–375
*PInvoke*. *See* Platform Invocation Services, .NET
platform, Microsoft .NET, 43–44. *See also* Microsoft .NET Framework
Platform Invocation Services, .NET, 365–371
declaring exported DLL functions, 366–369
marshalling and, 370–371
using callback functions, 369–370
pointer operators, *372*
pointers, 371–373, *372*
polymorphism, 37–40
virtual methods and, 134–140
portability, 273
portable executable files (PEs), 48, 347, 352, 389. *See also* assemblies
positional parameters, 173–174
postfix increment and decrement operators, 225–226
precedence, operator. *See* operator precedence
preemptive multitasking, 324

prefixes
  increment and decrement operators, 225–226
  name, 71–72, 74
primary expression operators, 213–218
  *checked* and *unchecked* operators, 218
  *sizeof* operator, 216–218
  *typeof* operator, 214–216
primitive types, 23, 77
priority classes, process, 334
priority levels, thread, 330–334
*Priority* property, 330
*private* access modifier, 31, 88, *91,* 121, 140
private assemblies, 391–392
problem domains, 25, 61, 73
procedural languages, 24
*Process* class, 354
processes, threads and, 326–327, 334
product information, 391
program flow control. *See* flow control statements
programming. *See* C# programming
*Programming Windows* (Petzold), 71
properties, 143–150
  accessor methods and, 143–144
  arrays and (*see* indexers)
  as class members, 90–91
  as smart fields, 143–144, 150
  compiler output and, 146–147
  defining and using, 144–145
  inheriting, 148
  lazy initialization and, 149–150
  read-only, 148
  *StackTrace* property, 279–280
  wrapping delegates in, 307–309
*protected* access modifier, 28, 31, 88, *91,* 121, 144
protected fields, 143–144
*public* access modifier, 28, 31, 61, 88, *91*. *See also* public members
*PublicKeyToken* value, 396

public members
  interfaces, 195
  objects and, 30, 31, 33
  overloaded operators, 288
publish/subscribe event design pattern, 90, 315

## Q

QFEs (Quick Fix Engineering updates), 402
qualification. *See* explicit interface member name qualification
querying
  about array rank, 154–155
  about attributes (*see* querying about attributes)
  for interface implementation by using *as,* 190–193
  for interface implementation by using *is,* 186–190
  metadata (*see* reflection)
  types, 349–351
querying about attributes, 166–172
  class attributes, 166–168
  field attributes, 171–172
  method attributes, 169–170
queues, 344
Quick Fix Engineering updates (QFEs), 402

## R

rank, array, 154–155
*Rank* property, 154–155
readability of code, 274–275
*ReadLine* method, 62
read-only fields, constants vs., 90, 102–105
*readonly* keyword, 90, 103–104
read-only properties, 148, 307
reference counting collection
  combining tracing collection and, 114
  deterministic finalization and, 108
  history of, 106
  performance and, 108–114
referenced assemblies, 390
reference types, 79–80

reflection, 347–363
assemblies and modules, 352–356
class attributes, 166–168
creating and executing code at run time, 360–363
field attributes, 171–172
late binding to COM components with, 383–385
late binding with, 357–359
metadata and, 50–51, *51,* 347
method, *80*
method attributes, 169–170
querying about attributes, 166–172
reflection API hierarchy, 347–348, *348*
*System.Type* class, 348–351
*typeof* operator and, 214–216
reflection API hierarchy, 166, 347–348, *348*
*ref* method parameters, 123–128
Registry
Assembly Cache Viewer and, 398
attributes and, 164–165, 173–174
COM components and, 377
.NET Framework and, 52
Visual Studio and, 56
relational operators, 226–229
resource management, 105–117
design pattern, 116–117
deterministic finalization, 107–108
history of, 106–107
just-in-time compilers and, 49
marking classes for deterministic finalization, 115–116
object termination and, 105–106
perfect solution to problem of, 114
performance and, 108–114
pinning and, 372, 373–375
RCWs and, 371
*Resume* method, 328
rethrowing exceptions, 268–269, 284–285
*ret* MSIL opcode, 69
retrieving *Type* objects
for instances, 348
from names, 349

return codes and error handling, 270–271
  exception handling vs., 271–273
*return* statements, 94, 263
return values. *See also* parameters
  delegate, 303
  exported DLL function, 370–371
  *Main* method, 61, 94
  overload operators, 288
  *void*, 263
reusability, 33, 35, 36, 375
revision numbers, 390, 399, 402
right associativity, 211–212
Roberts, Eric S., 259
round robin scheduling, 334
rules of substitutability, 117–118
running applications, 59–60
running code after catching exceptions, 269–270
run time
  checking type compatibility at, 188
  compilers and, 49
  creating and executing code at, 360–363
  environment, 45
  exceptions, 83–84, 286 (*see also* exception handling)
  metadata and, 50
  read-only fields and, 103
runtime callable wrappers (RCWs), 377–379, *378,* 381, 382
*rvalue* value, 221, 229, 375

## S

safe mode configuration files, 402–404
safe points, 330
safety, thread. *See* thread safety
safety, type, 79, 87–88
scheduling threads, 330–334
scope, namespaces and, 84
*sealed* classes, 120–121
security
  assemblies and, 352
  intranets and, 354
  manifests and, 391
  .NET Framework, 51–52

selecting COM interfaces with dynamic type discovery, 382–383
selection statements, 235–245
  *if* statement, 235–240
  *switch* statement, 240–245
self-describing components, 52, 163–164, 391
semicolon (;), 236
serialized data, 277
serializing code access, 335–339
services, unmanaged code using .NET, 53
*set* method (setter), 90, 144–145, 148
shallow copies, *81*
shared assemblies, 390, 392, 395–397, 399
shfusion.dll (Assembly Cache Viewer), 397, 398
signatures, assembly, 395
signed assembly hash, 390
Simonyi, Charles, 71
simple assignment operators, 229–232
simple statements, 236
single-dimensional arrays, 151–152
single-file assemblies, 390, 392–393
single-threaded apartments (STAs), 385–386
single-threaded applications, 323
single-use attributes, 178–179
singly rooted object hierarchy, 87
*sizeof* operator, 216–218
skeleton code, 63
*Sleep* method, 327–328
SmallTalk, 78
smart arrays, indexers as, 90, 157. *See also* indexers
smart fields, properties as, 90, 149–150. *See also* properties
SmartTags, .NET, 43
sn.exe (Strong Name tool), 395–397, 399–402
square brackets operator ([]), 164, 213, 292
*StackTrace* property, 279–280
Starbase CodeWright, 58
*Start* method, 326
STAs (single-threaded apartments), 385–386
*static* keyword, 61, 90, 104, 141, 288, 293, 366

static members
  access to, 142
  constructors, 95, 104
  delegates as, 305–307
  exported DLL functions, 366
  instance members vs., 97–99
  *Main* method as, 61
  methods as, 140–142
  overloaded operators, 288
  read-only fields, 104
  user-defined conversions, 293
stepping, *for* statement, 249
*stloc* MSIL opcode, 228
*StringBuilder* class, 370
*String* class, 237–238
string operators (+, +=), 210, 229
*String* values, 277
strong names, 395–397, 399–402
Strong Name tool (sn.exe), 395–397, 399–402
Stroustrup, Bjarne, 31, 96
*struct* keyword, 293
structured programming, 258
structures
  determining sizes of, 217–218
  objects vs., 25, 26–27
  user-defined conversions, 292–299
  using exported, 365, 366
*SubClassOf* method, 359
subscribing to events, 316, 318
substitutability, 37, 83, 117–118
*Suspend* method, 328
*switch* statement, 240–245
  combining *case* labels, 242–244
  fall-through and, 244–245
synchronization. *See* thread safety
syntactic sugar, operator overloading as, 287
syntax highlighting, 56, *57*
*System.Array* class, 150, 152, 154–155
*System.Attribute* class, 165
*System.Console* class, 60, 62
*System.Exception* class. *See Exception* class

*System.Int32* type, 78
*System.InvalidCastException,* 84, 382
*System.IO.Directory* class, 359
*System.Monitor* class, 335–339
*System* namespace, 62–63
*System.NullReferenceException,* 84
*System.Object* class, 72, 87, 384
*System.Reflection.Emit* namespace, 360–363
*System.Reflection* namespace, 347–348, *348*
*System.Runtime.InteropServices* namespace, 367, 371, 381
*System.ThreadAbortException,* 328
*System.Threading* namespace, 325
*System.Threading.Thread* class. *See Thread* class
*System.Type* class. *See Type* class

## T

targets, attribute, 176–177
technologies, .NET, 43–44. *See also* Microsoft .NET Framework
termination, 105
*Thinking in Java* (Eckel), 32
third-party editors, 58
*this* keyword, 99, 157–158
*ThreadAbortException,* 328
*Thread* class, 325–326, 327–330
  apartments and, 386
  creating threads and thread objects, 327
  destroying threads, 328–330
  managing lifetimes, 327–328
threading, 323–345
  *AppDomain* and, 326–327
  COM models, 385–386
  context switching and, 324–325
  creating threads and thread objects, 327
  destroying threads, 328–330
  example application creating secondary thread, 325–326
  garbage collection (*see* resource management)
  guidelines, 343–345
  managing thread lifetimes, 327–328
  multitasking and, 324
  scheduling threads, 330–334
  single-threaded vs. multithreaded applications, 323–324

*Thread* class, 327–330
thread safety and synchronization, 334–343 (*see also* thread safety)
when to avoid, 344–345
when to use, 343–344
*Thread* method, 330
*ThreadPriority* value, 330
thread safety, 334–343, 345
.NET classes and, 343
protecting code with *Monitor* class, 335–339
using *Mutex* class for synchronization, 341–343
using monitor locks with *lock* statement, 339–341
throwing exceptions, 267
rethrowing exceptions, 268–269, 284–285
*throw* keyword, 267, 268–269
tight cohesion, 273
tight coupling, 35
*Timeout.Infinite* value, 328
timeslices, 324, 325
*TimeSpan* value, 343
tlbimp.exe (Type Library Importer), 378
*ToString* method, *81,* 283, 298
tracing collection
combining reference counting collection and, 114
deterministic finalization and, 108
performance and, 108–114
tree view, ILDASM, 67, *68*
*true* value, 226, 235
*try* blocks, 267, 268
*catch* keyword and, 268
COM component errors and, 381
constructors and, 276
design issues, 283–285
*return* statements and, 263
*TypeBuilder* class, 361–362
type casting
boxing and unboxing and, 80
deterministic finalization and, 116
dynamic type discovery and, 382–383
interfaces and, 203
operator, 221
type system and, 82–84
upcasting, 37
user-defined conversions, 292–299

*Type* class, 348–351
attribute parameters and, 175
class attributes and, 168
COM component *ProgID* and, 383–384
COM interfaces and, 382
interrogating types, 349–351
method attributes and, 170
retrieving types from names, 349
retrieving types of instances, 348
*typeof* operator and, 214–216
type libraries (typelibs), COM, 50
generating metadata from, 377–379, *378, 379*
Type Library Importer (tlbimp.exe), 378
*typeof* operator, 168, 214–216
types
aliases and, 81–82, *81–82*
attribute parameter, 175
casting between (*see* type casting)
catching multiple exception, 280–281
classes as, 29
class member, 90–91
dynamic type discovery and COM components, 382–383
Java, 23
manifest data on, 390
marshalling exported DLL function parameters and, 370–371
reading COM component information, 377–379, *378, 379*
reflection and (*see* reflection)
*switch* statement and, 240
system (*see* Common Type System (CTS), .NET)
*Type* class (*see Type* class)
type safety, 79, 87–88
user-defined, 287
*Types* array, 352–355

## U

unary operators (+ ,-), 209, 219-221, 291
overloading, 288, 291
unboxing, 79–80
*unchecked* operator, 218
Unicode character set, 368–369, 371
unified type system. *See* Common Type System (CTS), .NET

uninstalling assemblies, 398
unmanaged code, 365–387
  COM interoperability and, 365, 375–386
  determination of, using ILDASM, 69
  interoperability and, 52–53, 365
  Platform Invocation Services and, 365–371
  unsafe code and, 365, 371–375
*UnmanagedType* enumeration, 371
unsafe code, 371–375
  *fixed* statement and, 373–375
  using pointers, 372–373, *372*
*unsafe* keyword, 216, 372
unscheduled fixes, 402
upcasting, 37, 40, 82–83. *See also* type casting
updates, assembly versioning and, 402–404
uppercase, 72
usage, attribute. *See AttributeUsage* attribute
usage, operator precedence, 212–213
user-defined conversions, 292–299
user-defined namespaces, 70
user-defined types, 287
user experience, .NET, 43
user issues, 32–33, 47, 299
usertype.dat file, 56–57
*using* directive, 62–63, 85–86, 326

## V

*validon* parameter, 176–177
value types, 78, 79–80, 216, 372
*value* variable, 145, 146–147
variable method parameters, 131–132
variables
  methods and, 28
  naming conventions, 72
  passing (*see* arguments; parameters; return values)
  pinning, 372, 373–375
  value types and, 78

versioning, assembly, 284, 352, 392, 399–404
creating safe mode configuration file, 402–403
example, 399–402
manifest data, 390
QFEs and default versioning policy, 402
using specific versions, 403–404
version numbers, 399
*virtual* keyword, 137, 148
virtual machine, 44
virtual methods, 132–140
interfaces and, 183
method overriding and, 132–134
polymorphism and, 134–140
Visual Basic, 23, 106–107, 112
Visual Basic.NET, 112
Visual C++, 56, 102
Visual SlickEdit, 58
Visual Studio 26, 56–57
Visual Studio.NET, 44, 50, 57
*void* return value, 61, 263, 303, 373

## W

*WaitOne* method, 343
*while* statement, 245–247
Win32 *Enum* functions, 369–370
Win32 *LPSTR* type, 370–371
Win32-portable executable files (PEs), 48, 347, 352, 389. *See also* assemblies
Win32 processes, 326–327
window handles, resource management and, 107
Windows. *See* Microsoft Windows
worker threads, 325–326
*WriteLine* method, 60, 62
writing applications, 55–60
choosing editors, 55–58
"Hello World" example, 58
running applications, 59–60
using command-line compiler, 58–59

## X

XML configuration files, 402–404