גירסה 2.01 – 24.3.2010

גירסה 2.00 – 21.7.2007

C# - מדריך למתכנתי ++C/C

מסמך זה הורד מהאתר http://www.underwar.co.il.

אין להפיץ מסמך זה במדיה כלשהי, ללא אישור מפורש מאת המחבר.

מחבר המסמך איננו אחראי לכל נזק, ישיר או עקיף, שיגרם עקב השימוש במידע המופיע במסמך, וכן לנכונות התוכן של הנושאים המופיעים במסמך. עם זאת, המחבר עשה את מירב המאמצים כדי לספק את המידע המדויק והמלא ביותר.

המסמך נכתב על ידי **ניר אדר** (nir@underwar.co.il). חלקים נכתבו ונערכו על ידי **שושן כהן** (psclil@gmail.com).

תודה ל**רותם גרוסמן** שבדק את המהדורה הראשונה של המסמך לפני הפצתו.

1. תוכן עניינים

1. תוכן עניינים 2

2. מבוא 5

3. תכנית ראשונה ותחביר בסיסי 7

3.1. HELLO, WORLD 7

3.2. תחביר בסיסי 8

4. משתנים 9

4.1. משתנים בשפת C# 9

4.2. תווים 13

4.3. CASTING 14

4.4. קבלת טווחי המשתנים 15

5. ערכים 16

6. אופרטורים ב-C# 17

6.1. אופרטורים של השמה 18

6.2. אופרטורים של השוואה 19

6.3. אופרטורים לוגיים 21

7. קלט / פלט 23

8. משפטי בקרה 26

8.1. IF_ELSE 26

8.2. WHILE 26

8.3. DO_WHILE 27

8.4. FOR 27

8.5. FOREACH 27

8.6. פקודות CONTINUE ו-BREAK 28

9. פונקציות 29

9.1. מבוא 29

9.2. העברת פרמטרים לפונקציה 31

9.3. פונקציות חופפות 34

10. מחלקות 35

10.1. מבוא 35

10.2. דוגמא ראשונה 35

10.3. הרשאות 39

10.4. פונקציות בונות 41
10.5. DEFAULT CONSTRUCTOR 42
10.6. פונקציות הורסות 42
10.7. PROPERTIES 44
10.8. משתנים ופונקציות סטטיים 47
10.9. העברת אובייקטים (משתני מחלקות) לפונקציות 50
10.10. מבנים והעברת פרמטרים לפונקציות 52

11. מחרוזות 53

11.1. מבוא ותחביר 53
11.2. STRINGBUILDER 56
11.3. מחרוזות ותווים מיוחדים 58

12. מערכים 59

12.1. הגדרות ודוגמאות ראשונות 59
12.2. העברת מערכים לפונקציה 60
12.3. פונקציה המחזירה מערך 60
12.4. מערכים רב ממדיים 63
12.4.1. מערכים מלבניים 63
12.4.2. מערך jagged 64
12.5. שיטות ומאפיינים של מערכים 65

13. מבנים תחביריים נוספים בשפה 68

13.1. מבנים 68
13.2. ENUMERATIONS 71
13.3. קבועים 73
13.4. READONLY 74
13.5. BOXING, UNBOXING 75
13.6. הוראות למהדר 76

14. הורשה 77

14.1. הורשה בשפת C# 77
14.2. דוגמא ראשונה 78
14.3. גישה אל שדות PRIVATE 78
14.4. אתחול מחלקת הבסיס 80
14.5. הרשאת PROTECTED 81
14.6. SEALED CLASS 82

15. פולימורפיזם 83

15.1. הצגת הצורך 83
15.2. מחלקות אבסטרקטיות ופונקציות טהורות 86
15.3. CASTING 89

15.4. ביצוע OVERRIDE לפונקציות של OBJECT 89

16. INTERFACE 91

16.1. NAMESPACE 91
16.2. INTERFACE 94
16.3. IENUMERATOR ,IENUMERABLE 96
16.4. ICOMPARABLE 99
16.5. INTERFACE הנגזר ממספר INTERFACES 100
16.6. שימוש בממשק IDISPOSABLE 101

17. מגנון בדיקת שגיאות - EXCEPTIONS 103

17.1. מבוא 103
17.2. APPLICATIONEXCEPTION 107
17.3. בניית EXCEPTION משלך 107

18. OPERATORS OVERLOADING 109

18.1. חפיפת אופרטורים בינאריים 109
18.2. INDEXER 113
18.3. פונקציות המרה 114

19. DELEGATES וטיפול באירועים 115

19.1. DELEGATE 115
19.2. טיפול באירועים 120

20. טיפים לניהול זיכרון חכם 123

20.1. האם ניהול הזיכרון לא מתבצע בשבילי? 123
20.2. הדורות השונים 123
20.3. ערימת האובייקטים הגדולים 125
20.4. מקטעי זיכרון 125
20.5. מה קורה בזמן איסוף זבל? 125
20.6. מספר טיפים 126
20.6.1. הימנעו מדור 1 126
20.6.2. הימנעו מקריאה ל-GC.COLLECT() 127
20.6.3. הימנעו מאובייקטים גדולים 127
20.6.4. הימנעו מפונקציות הורסות 128

21. סיום 129

2. מבוא

מסמך זה מציג את עיקרי שפת #C, עם התאמה לסביבת Net Framework 1.1.
מסמך זה איננו מדריך למתחילים. מסמך זה מניח שליטה בשפת ++C על בוריה ומתקדם על מנת להכיר למתכנתי ++C את השפה החדשה.

שפת #C הינה התשובה של מיקרוסופט לשפת Java הפופולארית. מבחינת תחביר שתי השפות מאוד קרובות, אולם ישנן הבדלים המייחדים כל אחת מן השפות. השפה הינה OOP מלאה, מאפשרת ירושה יחידה (בדומה ל-Java) ומכילה תמיכה בממשקים ובאירועים.

מה חדש בסביבת Net.?

- הקבצים הנכתבים ב-Net. אינם מתורגמים לשפת מכונה (אסמבלר), אלא לשפת ביניים – IL. בזמן הריצה, Net Framework. מבצעת את הקומפילציה האחרונה ואת ההרצה של התוכנית. גישה זו דומה לגישת קוד הביניים של שפת Java. המטרה בשתי השפות היא ליצור שפה שאינה תלוית מכונה. קוד Java יכול לראות על מגוון רחב של מחשבים ומערכות הפעלה, וסביבת Net. הינה צעד של חברת מיקרוסופט בכיוון דומה.
- Net. מכילה ספריית מחלקות עשירה ביותר בשם CLR שזהו קיצור של השם Common Language Runtime. כל פונקציות ה-Runtime של כל שפות התכנות השונות מאוחדות ב-Net. לספריה אחת מרכזית זו ובכך מושגות מספר מטרות חשובות:
  - השפות עשירות מאוד מבחינת המחלקות המגיעות ביחד עם השפה ומבחינת האפשרויות שהן מספקות למתכנת.
  - מתכנתים יכולים לעבור בקלות יחסית בין שפות בסביבת Net. – המחלקות והפונקציות בהם הם משתמשים נשארות מוכרות ומקלות על המעבר בין השפות.
- ASP.Net – טכנולוגיה חדשה לבניית אתרי אינטרנט, המציגה תפיסה חדשה איך צריך לבנות את האתרים, ומאפשרת שילוב של הכוח החזק של השפה עם סביבת הרשת.

מהי סביבת Net.?

סביבת Net. היא סביבת עבודה (frameworks) המספקת מחלקות שירות עימן השפה עובדת, מריצה את התוכניות שלנו בפועל ומספקת מצע עליו קמות התוכנות.

כאשר אנחנו מתכנתים בשפת #C אנחנו למעשה בונים תוכנית מעל הטכנולוגיה של הסביבה:

- התוכנית שנכתוב תשתמש במחלקות שהינן חלק מה-CLR של סביבת העבודה.
- הקוד שנכתוב יקומפל ל-IL. בעת הרצת הקוד סביבת Net. היא האחראית לקחת קוד זה ולהפכו לשפת מכונה.
- סביבת העבודה מגדירה את מנגנון ניהול הזיכרון האוטומטי (Garbage Collector), את כמות הזיכרון שמשתנים בשפות השונות תופסים ופרמטרים רבים נוספים הקשורים להרצה בפועל של התוכנית.

סביבת Net. מכילה יותר מאשר את שפת #C. בנוסף ל-#C, סביבת Net. מכילה שפות נוספות כגון ++C ו-VB.Net המשתמשות אף הן בסביבת העבודה על מנת לרוץ.

3. תכנית ראשונה ותחביר בסיסי

3.1. Hello, World

הדוגמא הקלאסית איתה מתחילים לרוב היא התוכנית המדפיסה על המסך את הצירוף "Hello World". דרך התוכנית הפשוטה הזו נוכל לראות את המרכיבים הבסיסים ביותר של השפה ולהתוודע ליסודות שלה.

```
using System;
public class CHelloWorld
{
      public static int Main(string[] args)
      {
            Console.WriteLine("Hello, World!");
            return 0;
      }
}
```

דגשים עליהם אנו למדים מהתוכנית:

- **ב-#C כל הפונקציות שייכות למחלקה כלשהי, כולל פונקציית ה-()Main.**
  הפונקציה ()Main היא נקודת ההתחלה של תוכנית ב-#C. הפונקציה מתחילה ב-M גדולה (Capital Letter) ושאר האותיות קטנות, והיא מוגדרת כפונקציה סטאטית.
  - הפונקציה הסטאטית Main צריכה להיות יחידה במחלקות השונות בתוכנית, על מנת שסביבת העבודה תמצא את נקודת ההתחלה של התוכנית.
  - הפונקציה מקבלת מערך של מחרוזות, שהינן פרמטרים המגיעים לתוכנית משורת ההפעלה. כל פרמטר נמצא בתא נפרד במערך.
  - בדומה לשפת ++C/C הפונקציה Main מחזירה 0 בסיום על מנת להודיע למערכת ההפעלה על סיום מוצלח של התוכנית.

- **הפונקציה ()WriteLine משמשת אותנו על מנת להוציא פלט.**
  בתוך התוכנית ישנה קריאה לפונקציה ()WriteLine שכותבת אל הפלט את המחרוזת "Hello, World" ויורדת שורה. עוד מהאפשרויות של פונקציה זו נציג בהמשך מסמך זה.
- **הפונקציות השונות בשפה משוייכות לאובייקטים ול-Namespaces.**
  הפונקציה ()WriteLine שייכת לאובייקט Console המוגדר ב- System Namespace.
  כדי להשתמש באובייקט זה, אנו רושמים בראש התוכנית - using System.
  ניתן לקרוא לאובייקט גם כך: ()System.Console.WriteLine.

3.2. תחביר בסיסי

התחביר בשפת #C דומה מאוד לתחביר בשפת ++C/C. השפה רגישה לשגיאות תחביריות, אולם סגנון הכתיבה עצמו הינו חופשי יחסית.

סגנון הכתיבה בשפת #C:

- **רווחים**: אין הקפדה על רווחים בין מילים, או על כך שכל הפקודה תהיה בשורה אחת, אולם מילים שמורות ושמות משתנים חייבים להיכתב ברצף ללא רווחים בין האותיות המרכיבות את המילה.
- **נקודה פסיק**: סוף כל פקודה מסתיימת בנקודה-פסיק (;).
- **הערות**: הוספת הערות לתוכנית נעשית על ידי הסימון // המשמש כהערה לשורה אחת, או על ידי הסימון /* */ בעזרתו ההערה יכולה להשתרע על מספר שורות.
- **בלוקים**: בלוק בשפת #C הינו קטע המוקף בסוגריים מסולסלות בתחילתו ובסופו. בלוק יכול להיות פונקציה, גוף תנאי if וכו'. הכתיבה המקובלת בשפת #C הינה מיקום הסוגריים המסולסלות בתחילת בלוק ובסופו בשורה נפרדת (בדומה לנהוג בשפת ++C ופחות בשפת C, בה נהוג לשים את הסוגריים המסולסלות הפותחות את הבלוק באותה שורה של תחילת הפונקציה / תנאי ה-if).

4. משתנים

4.1. משתנים בשפת #C

משתנה הוא תא/מספר תאים בזיכרון, להם אנו נותנים שם מיוחד שיזהה אותם ויכולים לשמור בהם מידע. סביבת Net., ובעקבותיה שפת #C מכילה מגוון של משתנים (המכונים גם – primitive types, בניגוד למופעים של מחלקות, הנקראים אובייקטים – objects).
סוגים שונים של משתנים מסוגלים להכיל מידע שונה. שפת #C כוללת משתנים שמיועדים לשמור מספרים שלמים, משתנים המיועדים לשמור תווים (אותיות), משתנים המיועדים לשמור מספרים רציונאליים, משתנים השומרים ערכים בוליאניים וכמו כן משתנים מורכבים יותר – Object וכן משתנה מחרוזת.

מבחינה תחבירית, ההצהרה על משתנים זהה להצהרה בשפות ++C/C והיא נראית כך:
Type <variable list>;

ניתן להגדיר משתנים בתור פונקציות, בתוך מחלקות, ובתור פרמטרים לפונקציות. לא ניתן להגדיר משתנים מחוץ למחלקות. ניתן להגדיר משתנים בכל חלק של הפונקציה, וכמו כן ניתן לאתחל משתנים מיד עם ההצהרה עליהם.

מכיוון שסביבת העבודה של #C הינה Net., כל השפות ב-Net. חולקות את אותם סוגים בסיסיים., אולם לסוגים שמות מעט שונים בשפות שונות.

הטבלה בעמוד הבא מציגה את המשתנים הבסיסיים בסביבת Net. העמודה השמאלית ביותר מציגה את השם בו המשתנה מוכר על ידי סביבת Net. עצמה (עם הזמן יתכנו מקרים בו יתקל המתכנת בשפות בשמות אלו בשמם הכללי ולא בזה של שפת #C). בעמודות נוספות מוצגים שמות סוגי המשתנים כפי שהם מוגדרים בשפת #C ובשפת C++.Net.

.Net Class	C#	C++
System.Byte	byte	char
System.SByte	sbyte	signed char
System.Int16	short	short
System.Int32	int	int or long
System.Int64	long	__int64
System.Uint16	ushort	unsigned short
System.Uint32	uint	unsigned int or unsigned long
System.Uint64	ulong	unsigned __int64
System.Single	float	float
System.Double	double	double
System.Object	object	Object*
System.Char	char	__wchar_t
System.String	string	String*
System.Boolean	bool	bool

סביבת .Net מציגה חידוש מבחינת המשתנים לעומת שפות שקדמו לה. בשפות C/C++ גודל המשתנים אינו מוגדר בבתים ומשתנה ממערכת הפעלה אחת לשנייה.

בסביבת .Net לעומת זאת כל המשתנים הם בעלי גודל מוגדר אותו מיד נציג. לכאורה מדובר על צעד אחורה מבחינת השפה, אולם למעשה זהו צעד חשוב ביותר.

העיקרון המנחה את שפת C# הוא ביטחון ומניעת אפשרויות לבאגים. בשפות בהן גודל המשתנים משתנה קורים מקרים רבים בהם התוכנית רצה כראוי על מחשב אחד, אולם במעבר למערכת אחרת בה גודל המשתנים שונה מתעוררות בעיות שונות עקב מתכנתים שהסתמכו על גודל משתנה מסוים.

על מנת לפתור זאת, שפת C# מגדירה באופן יחיד מה גודל המשתנה, ומסירה באמצעי פשוט זה את הערפל מעולם המשתנים.

הטבלה הבאה מציגה את גודלם של המשתנים השונים בשפת #C:

Variable	Size (in bits)	Range
sbyte	8	-128 to 127
short	16	-32768 to 32767
int	32	-2^{31} to 2^{31}
long	64	-2^{63} to 2^{63}
byte	8	0 to 2^{8}
ushort	16	0 to 2^{16}
uint	32	0 to 2^{32}
ulong	64	0 to 2^{64}
float	32	$\pm 1.5 \times 10^{-45}$ to $\pm 3.4 \times 10^{38}$
double	64	$\pm 5.0 \times 10^{-324}$ to 1.7×10^{308}
char	16	Unicode chars
bool	8	true or false

כחלק מרעיון הבטחון של השפה, שפת #C מפרידה לחלוטין בין ביטוי לוגי לביטוי מתמטי.
כלומר, בשפה לא ניתן להתייחס לערך 0 כאל false ואל 1 כאל true.
איך שינוי זה משפיע עלינו? נביט למשל בקוד הבא:

```
if (a = 3) b = 5;
```

אם קוד זה היה נכתב בשפת C, הוא היה עובר קומפילציה, ו-b תמיד היה מקבל את הערך 5. זאת מכיוון שההשמה a = 3 שמה במשתנה a את הערך 3, ו-3 מוגדר בשפת C כערך אמת. באגים רבים הופיעו בשפה באופן דומה. שפת #C מונעת בעיה זו על ידי ההפרדה בין הביטויים הלוגיים לביטויים המתמטיים. הקוד כפי שהוצג לעיל לא יעבור הידור בשפת #C.

דוגמא להגדרת משתנים בתוכנית והדפסתם

הדוגמא הבאה מציגה שימוש בסיסי ביותר במשתנים: אנחנו מגדירים מספר משתנים, מציבים לתוכם ערך בעת האתחול, ומיד לאחר מכן מדפיסים את ערכם ומסיימים את התוכנית.

```
using System;

public class CVariableExample1
{
	public static int Main(string[] args)
	{
		int x = 5, y = 10;
		double dValue = 12.34;
		Console.WriteLine("Variables: ({0}, {1}, {2})",
			x, y, dValue);
		return 0;
	}
}
```

פלט התוכנית יהיה:

```
Variables: (5, 10, 12.34)
```

פקודת הפלט מתנהגת בדומה לפקודת printf בשפת C. המחרוזת {0} הינה התייחסות אל המשתנה הראשון שמגיע לאחר מכן ברשימת המשתנים, {1} היא התייחסות אל המשתנה השני, וכו'.

4.2. תווים

תו הינו משתנה המסוגל לקבל ערך Unicode כלשהו. לרוב תווים משמשים על מנת להכיל אותיות או סימנים.

בניגוד לתווים בשפת C/C++ תווים אינם מסוגלים לקבל ערכים מספרים אלא רק ערכים שהם מסוג ערך Unicode. השורה הבאה <u>לא</u> תעבור הידור בשפת C#:

```
char ch = 48; // bad
```

אופן ההשמה למשתנה char נשאר דומה לשפות קודמות. על מנת להזין ערך של תו כלשהו לתוך משתנה, נעטוף אותו במרכאות יחידות, לדוגמא:

```
char ch = 'A';
char ch2 = '!';
```

ניתן לבצע השמה בין משתנה char אחד למשתנה char אחר בצורה דומה לזו של שפת C/C++:

```
char ch = 'a', ch2;
ch2 = ch;
```

בתום שורות אלו ch2 יכיל את הערך של המשתנה ch2, כלומר את הערך 'a'.

4.3. Casting

ניתן להכריח משתנה או ביטוי להיות מסוג מסוים על ידי פעולת המרה (Casting).
הצורה הכללית נראית כך: (type)expression

דוגמא:

```
using System;

public class CCastingExample
{
	public static int Main(string[] args)
	{
		int a = 12, b = 5;
		double c = a / b;
		double d = (double)a / b;
		Console.WriteLine("c = {0}, d = {1}", c, d);
		return 0;
	}
}
```

כאשר ביצענו את הפעולה a / b והצבנו אותה ב-c, הפעולה בוצעה על שלמים, וב-c הוצב בסופו של דבר 2. לעומת זאת, במקרה של d, קודם ביצענו המרה של a ל-double, ולכן הפעולה בוצעה על מספרים ממשיים, וב-d הוצב 2.4.

הערה חשובה: ב-C/C++ ניתן לבצע השמה ממשתנה גדול יותר אל משתנה קטן יותר, למשל השמה מ-long אל int, תוך איבוד הספרות הנוספות. בשפת C# הוחלט כי הדבר אסור. על ידי נקיטה בגישה זו - שפת C# מחמירה יותר מהשפות הישנות, שוב עקב הביטחון שהיא רוצה להביא למתכנת. שפת C# מצהירה למעשה שלא יהיה מקרה שנאבד בו מידע עקב השמה בלי שנהיה מודעים לכך.

נביט בקוד הבא:

```
double f = 3.23;
int i = f;
```

קוד זה לא יעבור קומפילציה מכיוון שיש בו איבוד מידע. על מנת שהקוד יתקמפל, יש צורך בהמרה מפורשת:

```
double f = 3.23;
int i = (int)f;
```

4.4. קבלת טווחי המשתנים

הטיפוסים הבסיסיים בשפת C# הם יותר מאשר סוגים, והם מכילים גם מאפיינים מסוימים של מחלקות ואובייקטים. לכל משתנה שניצור מוגדרות מספר שיטות בהן ניתן להשתמש, וכן ישנן גם פונקציות השייכות לטיפוסים עצמם (פונקציות סטאטיות). דוגמא ראשונה לכך נראה מיד, התוכנית הבאה מדגימה קבלה של הערכים המקסימאליים והמינימאליים הניתנים להשמה במשתנים מסוג int או long. בצורה דומה ניתן לקבל את הערכים עבור שאר סוגי המשתנים.

```
using System;

public class CVariableExample2
{
	public static int Main(string[] args)
	{
		Console.WriteLine("MaxInt = {0}, MaxLong = {1}",
			int.MaxValue, long.MaxValue);
		Console.WriteLine("MinInt = {0}, MinLong = {1}",
			int.MinValue, long.MinValue);
		return 0;
	}
}
```

יש לשים לב כי לא לכל מחלקה יש את המאפיינים minValue/maxValue. מאפיינים אלה קיימים באופן אוטומטי בשפה רב עבור הטיפוסים הבסיסיים.

5. ערכים

<u>עבודה בבסיס אוקטלי והקסדצימלי</u>

ניתן לייצג קבועים בבסיס אוקטלי ובבסיס הקסדצימלי בשפה. הוספת הקידומת 0x תגרום למהדר להתייחס לקבוע כאל קבוע בבסיס הקסדצימלי. הוספת הקידומת 0 תגרום למהדר להתייחס לקבוע כקבוע בבסיס אוקטלי.

<u>תווים מיוחדים</u>

הטבלה הבאה מסכמת תווים מיוחדים, שנכתבים על ידי הוספת הסימן \ לפניהם:

Escape Sequence	Represents
\a	Bell (alert)
\b	Backspace
\f	Formfeed
\n	New line
\r	Carriage return
\t	Horizontal tab
\v	Vertical tab
\'	Single quotation mark
\"	Double quotation mark
\\	Backslash
\?	Literal question mark
ooo	ASCII character in octal notation
\x*hh*	ASCII character in hexadecimal notation
\x*hhhh*	Unicode character in hexadecimal notation if this escape sequence is used in a wide-character constant or a Unicode string literal. For example, `WCHAR f = L'\x4e00'` or `WCHAR b[] = L"The Chinese character for one is \x4e00".`

6. אופרטורים ב-C#

אופרטורים הם פעולות שניתן לבצע על ערכים, אובייקטים ומשתנים בשפה. האופרטורים הקיימים בשפה הוגדרו על ידי בוני השפה ובוני אוסף המחלקות.

אוסף האופרטורים הקיימים בשפה:

<table>
<tr><th>Operator category</th><th>Operators</th></tr>
<tr><td>Arithmetic</td><td>+ - * / %</td></tr>
<tr><td>Logical (boolean and bitwise)</td><td>& | ^ ! ~ && || true false</td></tr>
<tr><td>String concatenation</td><td>+</td></tr>
<tr><td>Increment, decrement</td><td>++ --</td></tr>
<tr><td>Shift</td><td><< >></td></tr>
<tr><td>Relational</td><td>== != < > <= >=</td></tr>
<tr><td>Assignment</td><td>= += -= *= /= %= &= |= ^= <<= >>=</td></tr>
<tr><td>Member access</td><td>.</td></tr>
<tr><td>Indexing</td><td>[]</td></tr>
<tr><td>Cast</td><td>()</td></tr>
<tr><td>Conditional</td><td>?:</td></tr>
<tr><td>Delegate concatenation and removal</td><td>+ -</td></tr>
<tr><td>Object creation</td><td>New</td></tr>
<tr><td>Type information</td><td>is sizeof typeof</td></tr>
<tr><td>Overflow exception control</td><td>checked unchecked</td></tr>
<tr><td>Indirection and Address</td><td>* -> [] &</td></tr>
</table>

נציג כעת בהרחבה חלק מן האופרטורים.

6.1. אופרטורים של השמה

אופרטור ההשמה = הוצג כבר על ידי מספר דוגמאות. אופרטור זה מציב לתוך משתנה את הערך המועבר בצד הימני של הביטוי. בדומה לשפות מודרניות אחרות, שפת C# מאפשרת לנו אופרטורים נוספים של השמה, לצורך קיצור ולצורך נוחות.

האופרטור +=. נביט בקוד הבא:

```
x += 7;
```

המשמעות של הפקודה: הוסף לתוכן של x את המספר 7. השורה הבאה זהה בפעולתה לשורה הבאה:

```
x = x + 7;
```

באופן דומה, האופרטור -= משמעותו לחסר מהמשתנה את הביטוי שבצד ימין של האופרטור.

<u>האופרטורים ++/--</u>

אופרטורים אלו פועלים על משתנה בודד. האופרטור ++ מגדיל את ערכו של המשתנה ב-1, ואילו האופרטור -- מקטין את ערכו של המשתנה ב-1. אם האופרטור ייכתב משמאל למשתנה עלו הוא פועל, הערך של המשתנה ישתנה לפני ביצוע הפקודה הנוכחת. אם האופרטור ייכתב מימין, ערך המשתנה ישתנה לאחר ביצוע הפקודה הנוכחית.

לדוגמא:

```
int a = 2, b = 2, c, d;
c = a++;
d = ++b;
```

קטע הקוד לעיל זהה (מבחינה לוגית) לקטע הקוד הבא מבחינת פעולתו:

```
int a = 2, b = 2, c, d;
c = a;
a = a + 1;
b = b + 1;
d = b;
```

כמו שניתן לשים לב – לא שונו אופרטורים אלו בשפת #C. העבודה איתם תהיה בדיוק כמו ב-++C/C.

6.2. אופרטורים של השוואה

לכל ביטוי בשפת #C יש ערך. בניגוד לשפות ישנות, קיימת בשפה הבדלה ברורה בין ביטויים לוגיים לבין ביטויים מתמטיים.

בשפת C:

- אם ערך הביטוי הוא 0, אזי הערך הלוגי המתאים לו הוא שקר (FALSE).
- עבור כל ביטוי השונה מ-0, הערך הלוגי המתאים לו הוא אמת (TRUE).

למשל, ערכו של הביטוי 5 הוא אמת. אם נגדיר משתנה מסוג int בשם x, ונשים בו 0, אזי ערכו של הביטוי x הוא שקר.

בשפת #C:

- true, false הינם ערכים לוגיים.
- ביטוי לוגי הינו אחד מהבאים:
  - ערך לוגי
  - תוצאה של ביטוי המכיל אופרטור השוואה.
  - אופרטורים לוגיים המופעלים על ביטויים לוגיים.

למשל הביטוי (x==3) הינו ביטוי לוגי שערכו true או false.

שפת #C מכילה מגוון אופרטורים לצורך השוואה. ניתן ליצור ביטויים לוגיים הכוללים אופרטורים אלו.

אופרטורים של השוואה בשפת #C:

אופרטור	משמעות
<	קטן
>	גדול
<=	קטן שווה
>=	גדול שווה
==	שווה
!=	לא שווה

6.3. *אופרטורים לוגיים*

מספר אופרטורים לוגיים (אונרים ובינאריים):

אופרטור	משמעות
!	NOT (אופרטור אונרי)
&&	AND לוגי (אופרטור בינארי)
||	OR לוגי (אופרטור בינארי)

אופרטור אונרי הוא אופרטור שפועל על ביטוי אחד. אופרטור בינארי הוא אופרטור שפועל בין שני ביטויים.

עבור האופרטור האונרי ! (משמעותו not), מתקיימת טבלת האמת הבאה:

!x	x
true	false
false	true

טבלת האמת של האופרטורים הבינאריים:

x	**Y**	**x && y**	**x || y**
false	false	false	false
false	true	false	true
true	false	false	true
true	true	true	true

- האופרטור && משמעותו AND (וגם).
- האופרטור || משמעותו OR (או).

הדגש החשוב ביותר כשאנחנו מסתכלים על האופרטורים הלוגיים הוא לזכור את ההבדל מבחינת הסוג המועבר אליהם. שפת C# מאפשרת אופרטורים לוגיים רק בין ביטויים לוגיים ולא בין ביטויים מתמטיים.

7. קלט / פלט

קלט ופלט בשפת #C נעשים בעזרת הפונקציות WriteLine ו-ReadLine של המחלקה **Console**. מחלקה זו אחראית לפעולות הקלט והפלט של שפת #C.
מלבד הדפסה של נתונים על המסך, הפונקציה מאפשרת לנו לעצב במידה מסוימת את הטקסט היוצא אל המסך. המחלקה מאפשר לנו לדאוג ליישור הפלט, להגדרת מרווחים קבועים בין פלטים, ופעולות עימוד נוספות.
לפני שנציג את השימוש במחלקה נעלה את השאלה: מדוע בעולם של היום, בו כל המערכות הינן חלונאיות, יש צורך בכלל במצב עבודה של שורת פקודה, ושימוש באמצעים "פרימיטיביים" כמו קלט/פלט בסיסיים?
התשובה קשורה לייעוד של התוכנית שאנו כותבים. במידה ואנחנו כותבים תוכנית שבני אדם ישתמשו בה, נשתמש לרוב בממשק חלונאי (הממשק החלונאי של Net.,. אוסף המחלקות System.Windows.Forms, לא יוצג במסמך זה). לעומת זאת, תוכנות רבות נכתבות על מנת להיות בשימושן של תוכנות אחרות. למשל: תוכנה אחת שואבת נתונים ממקור נתונים מרוחק, ומביאה אותן אל התוכנה השנייה לניתוח בפורמט הנוח ביותר האפשרי. במקרים כאלה מתעורר הצורך בשימוש באמצעי קלט/פלט דרך ה-Console, לצורך העברת קלטים ופלטים בין התוכניות.
בהתאם לצורך זה – מגנון הקלט/פלט ב-#C ניתן לשימוש בני אדם, אולם כותבי השפה לא תכננו שתוכנות גדולות יכתבו בו. לפיכך – בגרסה 1.1 של הסביבה חלק מאפשרויות השליטה שבעבר היו קיימות על ה-Console לא נכללו בשפה, לדוגמא: ניתן להציג טקסט בצבעים שונים, אולם אנחנו מוגבלים למספר קטן של צבעים. כמו כן, לא ניתן לגרום לסמן שמצביע היכן הטקסט הבא יוזן לקפוץ לכל נקודה על המסך. ("gotoxy").
עם זאת, כאמור, ישנן אפשרויות לעיצוב הטקסט על מנת שיהיה ברור ותבניתי יותר, אותם נציג מיד.

כפי שכבר ראינו בדוגמאות קודמות, על מנת להציג משתנה בתוך השיטה WriteLine, יש לשים בסוגריים מסולסלות את מספרו הסידורי ברשימת הפרמטרים (כאשר הפרמטר הראשון הינו בעל המספר הסידורי 0), אותה אנו מעבירים אל WriteLine.

ניתן לדאוג גם ל**יישור המשתנה לפי עמודות**, וקביעת מרווח קבוע להצגת המשתנה. נעשה זאת על ידי הוספת פסיק לסוגריים המסולסלות, ורישום גודל הרווח הרצוי. מספר חיובי יאמר יישור לימין, בעוד מספר שלילי משמעותו יישור לשמאל.

דוגמא:

```
using System;

public class CConsoleExample1
{
	public static int Main(string[] args)
	{
		int x = 5, y = 10, z = 20;
		Console.WriteLine("{0, -10}{1, -10}{2, -10}", x, y, z);
		return 0;
	}
}
```

פלט התוכנית יהיה:

```
5         10        20
```

אם נשנה את השורה להיות:

```
Console.WriteLine("{0, 10}{1, 10}{2, 10}", x, y, z);
```

הפלט יהיה:

```
         5        10        20
```

לקורא מומלץ לנסות דוגמא בעצמו המכילה מספר שורות, ולראות יותר טוב כיצד היישור פועל.

הצגת המספר בבסיס שונה: ניתן לקבוע אם מספר יוצג בבסיס מסוים, על ידי הוספת נקודתיים ואחת מהאותיות בטבלה הבאה:

אות	משמעות
D או d	הצגת המספר בבסיס 10.
E או e	הצגת המספר ברישום מעריכי
F או f	הצגת מספר ממשים
X או x	מציג מספר בבסיס הקסדצימלי. אם x קטן האותיות יהיו קטנות, ואם X גדול האותיות יהיו גדולות.

דוגמא: נציג את המספרים נציג את המספרים העשרוניים 5 ו-10 בבסים הקסדצימלי.

```
using System;

public class CConsoleExample2
{
	public static int Main(string[] args)
	{
		int x = 5, y = 10;
		Console.WriteLine("{0, -10:X}{1, -10:X}", x, y);
		return 0;
	}
}
```

פלט התוכנית:

```
5         A
```

8. משפטי בקרה

רוב משפטי הבקרה בשפת C# זהים לחלוטין לאלו שבשפת C/C++ מבחינת התחביר שלהם ומבחינת אופן השימוש בהם. שפת C# מציגה מבנה תחבירי חדש, שהגיע משפות הסקריפטים של יוניקס, והוא משפט הבקרה foreach.
שינוי חשוב הינו הערך אותם יכולים לקבל הביטויים בתור condition. בניגוד לשפות C/C++, בשפת C# הערך בביטוי חייב להיות ערך לוגי ולא ערך מתמטי.

אנו מניחים כי הקורא שולט בשפת C/C++, ולכן בדפים הבאים רק נזכיר את התחביר של משפטי הבקרה השונים.

8.1. *if_else*

```
if (condition) statement;

if (condition)
{
       // block
}

if (condition) Statement1;
else Statement2;
```

8.2. *while*

```
while (condition) statement;

while (condition)
{
       // block
}
```

8.3. *do_while*

```
do
{
      // block
} while (condition);
```

8.4. *for*

```
for (initialization; condition; increment)
      statement;
```

8.5. *foreach*

לולאה חדשה זו מאפשרת לעבור על כל האיברים במערך או ב-collection. דוגמא:

```
using System;

public class CForEachExample1
{
      public static int Main(string[] args)
      {
            int[] vector = new int[]{1, 2, 3, 4, 5};
            foreach (int iItem in vector)
            {
                  Console.WriteLine ("Current Item is {0}", iItem);
            }
            return 0;
      }
}
```

קטע קוד זה ידפיס על המסך את המספרים 1 עד 5. בדוגמא הצגנו שימוש במערך, בו נתעמק יותר בהמשך, וכעת נסביר רק שזו הדרך הסטנדרטית להגדיר מערך בשפה.

8.6. פקודות continue ו-break

break, continue הן שתי מילים שמורות, המאפשרות שליטה על ביצוע הלולאה.
מינוח מקובל: מעבר אחד על גוף הלולאה - מתחילת הסוגריים המסולסלות ועד סופן, נקרא **איטרציה**.
המילה השמורה continue, כאשר אנו משתמשים בה בתוך לולאה, אומרת לשפת C# לסיים את האיטרציה הנוכחית של הלולאה, ולעבור אל האיטרציה הבאה.
המילה השמורה break אומרת ל-C# לסיים את הלולאה הנוכחית באופן מיידי, ולעבור לקוד שאחרי הלולאה. לדוגמא:

```
using System;

public class CForEachExample2
{
	public static int Main(string[] args)
	{
		int[] vector = new int[]{1, 2, 3, 4, 5};
		foreach (int iItem in vector)
		{
			if (iItem == 3) continue;
			Console.WriteLine ("Current Item is {0}", iItem);
		}
		return 0;
	}
}
```

יודפסו על המסך כל המספרים פרט למספר 3, עליו התוכנית מדלגת.

9. פונקציות

9.1. מבוא

פונקציה הינה מקטע קוד שלם, שמקבל פרמטרים, מבצע תהליך ומחזיר ערך כלשהו. בכדי להפעיל את הפונקציה יש לקרוא לה באמצעות כתיבת שמה. הפונקציה יכולה לקבל קלט (פרמטר אחד או יותר) אשר ישפיע על פעולתה. לדוגמא: הפונקציה Max שמקבלת כקלט מספר כלשהו של ערכים מספריים, ומחזירה את הגדול מביניהם.

בניגוד לשפות ++C/C, בשפת #C כל הפונקציות חייבות להיות שייכות למחלקה כלשהי.
שפת #C איננה כוללת "פונקציות גלובליות", פונקציות הנמצאות מחוץ לכל מחלקה.
במקומות בהם בשפות אחרות נדרשות פונקציות כאלו, שפת #C מציעה למתכנת להשתמש בפונקציה סטאטית של מחלקה כדי להשיג את ההתנהגות הרצויה.
בשפת #C אין הצהרה מראש על פונקציות. יש לממש פונקציה מיד עם כתיבת החתימה שלה.

הגדרה כללית של פונקציה:

```
[modifiers] return-type method-name([parameter-list])
```

חלקי ההגדרה השונים:

- **modifiers** מילים שמורות שמציינות הרשאה (public, private וכדו') או מאפיינים מיוחדים של הפונקציה (למשל – האם הפונקציה היא פונקציה סטאטית).
- **return-type** הערך אותו הפונקציה מחזירה.
- **mothod-name** שם הפונקציה
- **parameters-list** רשימת הפרמטרים שהפונקציה מקבלת, אם היא מקבלת.

דוגמא:

```
using System;

public class CFunctionExample1
{
	public static int Max(int x, int y)
	{
		return (x > y) ? x : y;
	}

	public static int Main(string[] args)
	{
		int x = 45, y = 100;
		Console.WriteLine("Max between {0} and {1} is {2}",
			x, y, Max(x,y));
		return 0;
	}
}
```

על פי ברירת המחדל, הערכים מועברים by value, כלומר אם נשנה את ערכי הפרמטרים בתוך הפונקציה, המשתנים המקוריים מחוץ לפונקציה יישארו ללא שינוי.

<u>פרמטרים לפונקציה:</u> פונקציה יכולה לקבל מספר כלשהו של פרמטרים. בדומה ל-C/C++, אנו מצהירים את סוג הפרמטר לפני כל פרמטר. פונקציה יכולה גם לא לקבל פרמטרים כלל.

<u>ערך מוחזר:</u> כל פונקציה יכולה להחזיר ערך אחד בלבד. חובה לרשום את הערך המוחזר של כל פונקציה.
פונקציה יכולה להחזיר void, כלומר, לא להחזיר דבר.

<u>משתנים פנימיים:</u> משתנים המוגדרים בתוך הפונקציה, הינם משתנים לוקליים של הפונקציה. טווח ההכרה שלהם הוא בתוך הפונקציה בלבד, ואילו אורך החיים שלהם הוא למשך הפונקציה בלבד.

9.2. העברת פרמטרים לפונקציה

על מנת להעביר משתנים לפונקציה, יש לכתוב בסוגריים העגולים שאחרי שם הפונקציה רשימה של משתנים, בפורמט הבא:

```
type1 name1, type2 name2, ... typeN nameN
```

כלומר, כותבים שם של טיפוס ואחר כך את שם המשתנה מאותו טיפוס. אם רוצים משתנה נוסף, מפרידים בפסיק בינו ובין המשתנה הקודם.

בזמן ריצת הפונקציה, כל המשתנים ששמותיהם נכתבו בשורה זו יהיו זמינים כאילו הוגדרו בתוך הפונקציה עצמה.
המשתנים הבסיסיים (int, double, long וכו') עוברים לפונקציות by value כברירת המחדל.
בהמשך כשנתעסק עם אובייקטים, נראה שהם עוברים by reference כברירת מחדל.

עם זאת, ברירת המחדל איננה חקוקה בסלע. ניתן להעביר פרמטרים בדרכים נוספות על ידי שימוש במילים שמורות של שפת C#:

העברה by ref: כאשר אנו מעבירים משתנה by reference, המשתנה המקורי מועבר לפונקציה, ולא העתק שלו. כתוצאה מכך, אם נשנה את הפרמטר במהלך הפונקציה, ישתנה גם המשתנה מחוץ לפונקציה.
העברה זו שימושית במיוחד כשאנחנו מדברים על מחלקות, אולם יהיו מקרים רבים בהם נרצה לשנות גם משתנים בסיסיים שמועברים לפונקציה. אחת הדוגמאות הנפוצות ביותר לכך היא הפונקציה swap הלוקחת שני ערכים ומחליפה את ערכם.

נציג מימוש של הפונקציה swap עבור משתנים שלמים בשפת C# תוך שימוש בהעברת המשתנים by reference.

קוד:

```
using System;

public class CFunctionExample2
{
	static void Swap(ref int x, ref int y)
	{
		int iTemp = x;
		x = y;
		y = iTemp;
	}

	public static int Main(string[] args)
	{
		int x = 45, y = 100;
		Console.WriteLine("Before: {0} , {1}", x, y);
		Swap(ref x, ref y);
		Console.WriteLine("After: {0} , {1}", x, y);
		return 0;
	}
}
```

במהלך ריצת התוכנית לאחר הקריאה לפונקציה Swap ערכי x ו-y בפונקציה Main הוחלפו.

פלט התוכנית:

```
Before: 45 , 100
After: 100 , 45
```

על מנת לבצע את ההעברה by reference היינו צריכים להשתמש במילה השמורה ref גם בקריאה לפונקציה Swap בפונקציה Main וגם ברשימת הפרמטרים של הפונקציה Swap. אם לא היינו עושים זאת, היינו מקבלים שגיאת קומפילציה, כלומר השפה בודקת פעמיים שאנחנו באמת מתכוונים להעביר את המשתנה בצורת התייחסות זו.

העברת פרמטרים בעזרת המילה השמורה out:

המילה השמורה out מאפשרת לנו להגדיר פרמטרים שינתנו לצורך החזרת ערכים לפונקציה הקוראת בלבד. משתני out מאפשרים לנו ליצור פונקציה שמחזירה יותר מערך אחד.

עבודה עם משתני out:

- אין צורך לאתחל את המשתנים הנשלחים כמשתני out.
- אין אפשרות להשתמש בתוך הפונקציה בערכי הפרמטרים המועברים, עד שהם מאותחלים במפורש בפונקציה.
- בסיום ריצת הפונקציה, הם חייבים להכיל ערך כלשהו.

בדוגמא הבאה אנו שולחים שני מספרים אל פונקציה, המחשבת את סכום מספרים אלו ואת מכפלתם, ומחזירה את התוצאה בעזרת משתנים שמועברים כ-out.

```
using System;

public class CFunctionExample3
{
	public static void DoMath(int a, int b, out int c, out int d)
	{
		c = a + b;
		d = a * b;
	}

	public static int Main(string[] args)
	{
		int a = 12, b = 4, c, d;
		DoMath(a, b, out c, out d);
		Console.WriteLine("Sum of {0} and {1} is {2}", a, b, c);
		Console.WriteLine("Mul of {0} and {1} is {2}", a, b, d);
		return 0;
	}
}
```

לאחר הקומפילציה אין הבדל בין ref ל-out, ההבדל הוא רק במגבלות על אתחול הפרמטרים.

9.3. פונקציות חופפות

ניתן להגדיר שתי פונקציות באותה מחלקה ובאותו שם, בתנאי שהפונקציות שונות בסוג הפרמטרים שלהן, במספר הפרמטרים או בצורה השליחה (value, ref, out). במקרה זה, בזמן הקריאה לפונקציה, הפרמטרים שישלחו הם אלו שיקבעו איזו פונקציה תופעל.

דוגמא:

```
using System;

public class CFunctionExample4
{
	public static void f1(double d)
	{
		Console.WriteLine("f1(double)");
	}

	public static void f1(int d)
	{
		Console.WriteLine("f1(int)");
	}

	public static int Main(string[] args)
	{
		f1(1);
		f1(1.1);
		return 0;
	}
}
```

בדוגמא זו, תקרא קודם כל הפונקציה f1 המקבלת int, ולאחריה זו המקבלת double.

במקרה וקריאה לפונקציה תוכל להתפרש כקריאה ל-2 פונקציות שונות והמהדר לא ידע לאיזו פונקציה הוא צריך להפעיל נקבל שגיאה בזמן ההידור שתתריע לנו על כך ותאפשר לנו לתקן את הקוד שלנו.

10. מחלקות

10.1. מבוא

מחלקה היא מבנה המאחד בתוכו פונקציות ומשתנים תחת שם אחד. המחלקה היא אחד המושגים הבסיסיים והחשובים ביותר בשפת #C, מעצם היותה שפה מכוונת עצמים באופן מובהק.
רעיון המחלקות הוא ניסיון לקרב את שפת התכנות למחשבה האנושית. אנו מגדירים עצמים ואוסף פעולות עליהם. נוכל למשל להגדיר עצם מסוג בן אדם. נגדיר (בעזרת שיטות) פעולות כגון הליכה, דיבור וכדומה. לאחר שהגדרנו אדם, נוכל להגדיר למשל גם "זמר", שיקבל את כל התכונות של העצם "אדם", ולהוסיף עליהן תכונות ופעולות נוספות, למשל, שירה.

מבחינה טכנית, המילה השמורה class פותחת את הגדרת המחלקה, לאחריה מופיע שם המחלקה, ולאחר מכן, בין סוגריים מסולסלים, מוגדרים המשתנים והפונקציות השייכות למחלקה.

```
class <class name>
{

}
```

הערה: הגדרת מחלקה ב-#C איננה מסתיימת בנקודה פסיק כפי שהיא מסתיימת ב-++C.

10.2. דוגמא ראשונה

בדוגמא מוגדרת מחלקה בשם CRectangle המייצגת מלבן. למחלקה שני משתנים פרטיים, ומספר פונקציות הפועלות על משתנים אלו.
התוכנית הראשית קובעת ערכים לצלעות המלבן ומדפיסה ערכים אלו על המסך.

דוגמא למחלקה:

```
using System;
public class CRectangle
{
	private int height, width;
	public void SetHeight(int newHeight) { height = newHeight; }
	public void SetWidth(int newWidth) { width = newWidth; }
	public int GetHeight() { return height; }
	public int GetWidth() { return width; }
}
public class CClassExample1
{
	public static int Main(string[] args)
	{
		CRectangle rec1 = new CRectangle();
		rec1.SetHeight(10);
		rec1.SetWidth(20);
		Console.WriteLine("Rectangle size: {0} x {1}",
			rec1.GetHeight(), rec1.GetWidth());
		return 0;
	}
}
```

דגשים בתוכנית:

- **הפונקציה Main**: בפונקציה ()Main, השייכת למחלקה CClassExample1, ממנה מתחילה התוכנית שלנו לרוץ, אנו מגדירים משתנה מסוג המחלקה CRectangle.
- **הצהרה על אובייקטים**: כאשר אנחנו מגדירים משתנה מסוג מחלקה (נכנה משתנה מסוג מחלקה בשם **אובייקט**) ב-#C, עדיין לא מוקצה עבורו זיכרון. למעשה יש לנו ביד משתנה התייחסות (reference) שאינו מצביע על דבר עם יצירתו, ויש צורך לבצע פעולה נוספת על מנת להקצות זיכרון בשבילו.

  אנו משתמשים באופרטור new על מנת להקצות זיכרון ולשים אותו במשתנה rec1.
- **ניהול זכרון אוטומטי**: אין צורך לשחרר את הזיכרון שהוקצה בתוכניות בשפת #C. הסביבה עצמה מזהה בעת צורך בעוד זכרון באילו משתנים איננו נשתמש יותר, ומשחררת אותם עבורנו, ניתן לסמן אובייקט ספציפי לתהליך זה ע"י השמת null באובייקט.
- **גישה לשדות**: כפי שכבר ראינו, כדי לגשת לשדות השונים של האובייקט שיצרנו אנחנו משתמשים באופרטור . (נקודה). לדוגמא: ()rec1.GetHeight.

דגשים לגבי המחלקה עצמה:

- **Getters/Setters**: במחלקה הוגדרו Getters ו-Setters כמו שמקובל בשפת C++.
בהמשך המסמך נראה את הדרך של שפת C# לממש Getters ו-Setters, שהיא שונה מזו שהוצגה בדוגמא ראשונית זאת, ומבוסס על Properties.
- **הרשאות**: ניתן לראות שההרשאות private ו-public קיימות גם בשפת C#. מיד נפנה להתעמקות רצינית יותר בהרשאות, אך בינתיים נשים לב כבר שבניגוד ל-C++ אנחנו מציינים את ההרשאה ליד כל משתנה או שיטה, ולא במרוכז כבר בשפת C++.

נביט בדוגמא הבאה:

הדוגמא מסתמכת על המחלקה CRectangle כפי שהוגדרה בדוגמא הקודמת:

```
public class CClassExample2
{
	public static int Main(string[] args)
	{
		CRectangle rec1, rec2;
		rec1 = new CRectangle();
		rec1.SetHeight(10);
		rec1.SetWidth(20);
		rec2 = rec1;
		Console.WriteLine("Rectangle size: {0} x {1}",
			rec2.GetHeight(), rec2.GetWidth());
		rec2.SetHeight(15);
		Console.WriteLine("Rectangle size: {0} x {1}",
			rec1.GetHeight(), rec1.GetWidth());
		return 0;
	}
}
```

עבור המשתנה rec2 לא יצרנו אובייקט חדש, אלא אמרנו כי rec2 שווה ל-rec1.
באמירה זו אמרנו למעשה כי rec2 ו-rec1 מתייחסים אל אותו אובייקט.
כאשר אנו משנים את אחד מהמשתנים, גם השני משתנה.

10.3. הרשאות

ב-C# קיימים אותם סוגי הרשאות כמו בשפת C++ - private, public, protected ובנוסף הרשאה חדשה – internal.

משמעות ההרשאות:

- **public**: פונקציה או משתנה המוגדרים public, ניתנים לגישה מכל מחלקה.
- **private**: פונקציה או משתנה המוגדרים private, ניתנים לגישה מפונקציות של אותה המחלקה בלבד.
- **protected**: פונקציה או משתנה המוגדרים protected, ניתנים לגישה מפונקציות של אותה המחלקה בלבד וכן מהמחלקות הנורשות. בהמשך נציג כיצד מתבצעת ירושת המחלקות בשפת C#.
- **internal**: פונקציה או משתנה המוגדרים internal ניתנים לגישה לכל פונקציה או מחלקה הנמצאת איתם באותו ה-namespace. הרשאה זו הינה הרשאת ברירת המחדל במידה ואיננו מציינים את ההרשאה הרצויה לנו, אך מומלץ ביותר להגדיר תמיד את ההרשאה בה אנו מעוניינים.

דוגמא להגבלה של הרשאת private: במחלקה CRectangle שהוצגה, אם היינו מנסים לכתוב את השורה הבאה בפונקציה ()Main, היינו מקבלים שגיאת קומפילציה, מכיוון ש-x הינו משתנה עם הרשאת private:

```
rec1.width = 4;
```

<u>רישום ההרשאות</u>: ב-C# רישום סוג ההרשאה הוא לפני כל פונקציה או משתנה, ומתקיים רק לגביו, וזאת בניגוד ל-C++ שם היה ניתן לרשום את סוג ההרשאה ועד להודעה חדשה הוא התקיים לגבי כל המשתנים והפונקציות שהוגדרו אחריו.

כמובן, לא ניתן להכריז על הרשאה בהכרזה הנותנת יותר הרשאות משל ההכרזה המכילה הכרזה מסויימת, הכרזה מסוג זה תביא לשגיאה בעת הקומפילציה של הקוד, לדוגמא: לא ניתן להכריז על מתודות או משתנים כ-public בתור מחלקה המוגדרת כ-private, ולא ניתן להכריז על שדות כ-public אם הם מסוג שהינו private.

<u>מוטיבציה – שימוש בהרשאות</u>

מדוע להשתמש בהרשאות ולא להגדיר את כל המשתנים כ-public? התשובה כרגיל היא בטחון. על ידי הרשאות נוכל להגביל את המשתמש, ולדאוג שבאובייקט יהיו רק ערכים חוקיים. למשל, נשכתב כעת המחלקה, כך שכאשר המשתמש בא לשנות את אורך או רוחב המלבן, השינוי יתקבל רק אם האורך או הרוחב החדשים יהיו גדולים ממש מ-0.

שיטת עבודה זו מאפשרת לנו לדאוג לכך שהאובייקטים שלנו יכילו ערכים חוקיים בכל רגע נתון.

ההרחבה:

```
public class CRectangle
{
	private int height, width;
	public void SetHeight(int newHeight)
	{
		if (newHeight > 0) height = newHeight;
	}

	public void SetWidth(int newWidth)
	{
		if (newWidth > 0) width = newWidth;
	}

	public int GetHeight() { return height; }
	public int GetWidth() { return width; }
}
```

10.4. פונקציות בונות

פונקציה בונה היא פונקציה הנקראת ברגע שאובייקט חדש נוצר, ותפקידה הוא אתחול האובייקט.

שם הפונקציה הוא כשם המחלקה, והיא איננה מחזירה אף משתנה.

פונקציה בונה יכולה לקבל פרמטרים או לא. כמו כן, ניתן להגדיר פונקציות בונות חופפות.

<u>דוגמא</u>

נוסיף למחלקה CRectangle שלנו פונקציה בונה שאיננה מקבלת פרמטרים, שתגדיר שכל מלבן חדש שנוצר יהיה בגודל 10 x 10.

```
public class CRectangle
{
	private int height, width;
	public void SetHeight(int newHeight)
	{
		if (newHeight > 0) height = newHeight;
	}

	public void SetWidth(int newWidth)
	{
		if (newWidth > 0) width = newWidth;
	}

	public int GetHeight() { return height; }
	public int GetWidth() { return width; }
	public CRectangle()
	{
		height = width = 10;
	}
}
```

10.5. Default Constructor

אם לא מגדירים עבור מחלקה כלשהי ב-#C פונקציה בונה, הקומפיילר מספק פונקציה בונה משלו כברירת מחדל. פונקציה זו, איננה מקבלת פרמטרים, ומאפסת את כל הערכים לאפס, את כל המשתנים הבוליאניים ל-false ואת כל האובייקטים ל-null.
אם מוגדרת פונקציה בונה כלשהי במחלקה, פונקצית ברירת המחדל לא נקראת.
נשים לב להבדל בין פונקציה זו לפונקצית ברירת המחדל ב-++C, שאיננה מאתחלת את המשתנים במחלקה. הבדל זה חשוב ביותר. <u>אנחנו יכולים להניח משתנים מאופסים</u>, ולא לאתחל אותם, במידה וזה המצב. עבור מחלקות עם הרבה נתונים, שימוש באיפוס האוטומטי והמנעות מאתחול נוסף הינה צעד חיוני על מנת לשפר את ביצועי התוכנית.

10.6. פונקציות הורסות

פונקציה הורסת מוגדרת על ידי שם המחלקה עם הסימן ~ לפני השם.
פונקציה הורסת איננה מקבלת פרמטרים ואינה מחזירה ערך, גם לא void. כמו כן, היא איננה יכולה להיות public.
ב-++C יש חשיבות גדולה מאוד לפונקציות ההורסות, מכיוון שידוע בדיוק מתי הן יפעלו. ב-#C לא ידוע מתי תופעל הפונקציה ההורסת, ולא בטוח שתפעל בכלל. כמו כן ניקוי הזיכרון, שהיה התפקיד המרכזי של הפונקציות ההורסות ב-++C, אינו קיים ב-#C, והוא נעשה באופן אוטומטי.
עקב עובדות אלו, חשיבות הפונקציה ההורסת ב-#C פחות, והשימוש בה הוא יותר נדיר.

דוגמא לפונקציה הורסת:

```
using System;

class MyClass
{
	public MyClass()
	{
		Console.WriteLine("MyClass Constructor");
	}

	~MyClass()
	{
		Console.WriteLine("MyClass Destructor");
	}
}

public class MainClass
{
	public static void Main()
	{
		MyClass m = new MyClass();
	}
}
```

פלט התוכנית:

```
MyClass Constructor
MyClass Destructor
```

נעיר שוב כי עבור רוב התוכניות שתכתבו בשפת C# לא סביר שתדרשו להשתמש בפונקציות הורסות. ה-GC שבא עם סביבת C# מצמצם באופן משמעותי ביותר את הצורך בפונקציות אלה.

<u>שימו לב ששימוש מיותר בפונקציות הורסות מאריך באופן גדול את אורך החיים של האובייקט שלכם ומגדיל את כמות הזיכרון בה משתמשת התכנית שלכם, ולכן משפיע כל הביצועים.</u>

Properties .10.7

ב-C# ניתן להגדיר משתנים ופונקציות כ-private או public. השימוש שהדגמנו בו הגדרנו משתנים כ-private, ולאחר מכן הגדרנו פונקציות get ו-set על מנת לגשת אליהן, הוא נפוץ ביותר, ולכן הוחלט להוסיף ל-C# מבנה דקדוקי מיוחד, המאפשר להאיץ תהליך זה.
מבנה זה הוא מנגנון ה-properties, שמאפשר לנו צורת עבודה המזכירה עבור משתמש הקצה עבודה מול שדות ישירות, כאשר בפועל יקראו פונקציות get ו-set עם כל קריאה של המשתמש.

נשפר שוב את המחלקה CRectangle, ונשתמש בתחביר החדש. נציג גם כיצד הפונקציה Main() משתנה.

```
using System;

public class CRectangle
{
	private int m_height, m_width;

	public int height
	{
		get
		{
			return m_height;
		}
		set
		{
			if (value > 0) m_height = value;
		}
	}
```

```
	public int width
	{
		get
		{
			return m_width;
		}
		set
		{
			if (value > 0) m_width = value;
		}
	}

	public CRectangle()
	{
		m_height = m_width = 10;
	}
}

public class CClassExample2
{
	public static int Main(string[] args)
	{
		CRectangle rec1, rec2;
		rec1 = new CRectangle();
		Console.WriteLine("Rectangle size: {0} x {1}",
			rec1.height, rec1.width);
		rec1.height = 10;
		rec1.width = 20;
		rec2 = rec1;
		Console.WriteLine("Rectangle size: {0} x {1}",
			rec2.height, rec2.width);
		rec2.height = 15;
		Console.WriteLine("Rectangle size: {0} x {1}",
			rec1.height, rec1.width);
		return 0;
	}
}
```

במחלקה CRectangle הגדרנו משתנה עם הרשאת private בשם m_height , וכן property בשם height. בהגדרת ה-property הגדרנו שני בלוקים, המשמשים כשתי פונקציות - get ו-set.

בפונקציה get החזרנו את המשתנה m_height, ובפונקציה set שינינו משתנה זה, אחרי שבדקנו את תקינות הקלט.

היתרון של השימוש ב-property הוא שכלפי מי שמשתמש באובייקט, ה-property מתנהג כמו משתנה, ועבור כותב המחלקה ה-property מתנהג כמו שתי פונקציה, כך שניתן לבצע בדיקות לפני שינוי הפרמטרים.

לאחר הקומפילציה של הקוד התכונות ממומשות כפי שממושו ב-++C, מתודת setPropertyName ומתודת GetPropertyName.

בנוסף חשוב לציין שמתכנתים רבים חוששים מהבדל מסויים בין ++C ו-#C והוא חוסר היכולת לקרוא למתודה בצורה inline (בצורה כזאת שבקוד יש קריאה לפוקנציה, אבל בפועל בקומפילציה המימוש מועתק לכל המופעים של הקריאה לצורך חסכון בתנועה במחסנית), ולכן חוששים מהשימוש במתודות ותכונות עבור שדות פשוטים, חשוב לציין עבור מתכנתים חסכניים אלו שבשפת #C כל תכונה שהביצוע שלה פשוט תהפוך לכזו לאחר תרגום קוד ה-IL לשפת מכונה (על ידי מנגנון JIT).

10.8. משתנים ופונקציות סטטיים

משתנים סטטיים אלו משתנים השייכים למחלקה ולא לאובייקט מסוים שלה, ולכן הם משותפים לכל האובייקטים של אותה מחלקה.

אנו יכולים להשתמש במשתנים סטטיים בתור המחליפים של המשתנים הגלובליים של שפת C/C++.

הגישה אל המשתנים הסטטיים נעשית בעזרת שם המחלקה.

ב-C++, כאשר היינו רוצים לאתחל קבועים השייכים למחלקה כלשהי, היינו מגדירים אותם בתור static, על מנת שיהיו שייכים למחלקה ולא לאובייקט. ב-C# זהו לא המצב – ישנה הפרדה בין המשתנים הסטטיים עליהם אנו דנים כעת, לבין קבועים של המחלקה המוגדרים באופן שונה שיוצג בהמשך. כאשר אנו מגדירים משתנים סטטיים, אנו מתכוונים לכך שהם יהיו משתנים של המחלקה.

נביט בדוגמא הבאה: נניח שאנחנו מממשים משחק מחשב, בו יש מבוך ובו מספר דלתות, ונניח כי נרצה מונה של כל הדלתות הקיימות במשחק. נוכל לממש מונה זה באמצעות משתנה סטטי, במחלקה "דלת":

```
using System;

public class CDoor
{
	private static int m_DoorCount;

	public CDoor()
	{
		m_DoorCount++;
	}

	public static int DoorCount
	{
		get { return m_DoorCount; }
	}
}
```

```
public class CStaticExample1
{
	public static int Main(string[] args)
	{
		CDoor door1, door2;
		door1 = new CDoor();
		door2 = new CDoor();
		Console.WriteLine("Number of doors is {0}",
			CDoor.DoorCount);
		return 0;
	}
}
```

נשים לב שכאשר רצינו לפנות ל-Property בשם DoorCount, כתבנו CDoor.DoorCount, כלומר את שם המחלקה, ולא שם של אחד מהאובייקטים שבה.

<u>אתחול משתנים סטטיים:</u>

- **ברירת מחדל**: כאשר אנו מגדירים משתנה סטטי במחלקה, ולא מאתחלים אותו, הוא מאותחל ל-0.
- **השמה מפורשת**: אתחול משתנים סטטיים בצורה מפורשת, מיד עם הגדרתם:

```
public class CStaticExample2
{
	public static int iVariable = 10;
}
```

אנו מאתחלים את המשתנה הסטטי מיד עם הגדרתו. המשתנה יאותחל פעם אחת בלבד במהלך ריצת התוכנית.

- **פונקציה בונה סטטית**: דרך נוספת לאתחל משתנים סטטיים היא על ידי פונקציה בונה סטטית. אנו יכולים להגדיר פונקציה בונה כפונקציה סטטית. פונקציה זו יכולה לאתחל את המשתנים הסטטיים של האובייקט, אך לא את המשתנים הרגילים של האובייקט. C# מבטיחה לנו כי הפונקציה הסטטית תיקרא לפני יצירת האובייקט הראשון של המחלקה. הפונקציה הבונה הסטטית אינה יכולה לקבל פרמטרים, אינה מחזירה ערך, ואף לא ניתן להגדיר אותה כ-public או כ-private.

נביט בדוגמא הבאה, שהיא שינוי של המחלקה CDoor שראינו קודם, המדגימה את הבנאי הסטטי.

```
using System;

public class CDoor
{
	private static int m_DoorCount;

	public CDoor()
	{
		m_DoorCount++;
		Console.WriteLine("In CDoor::Ctor");
	}
	static CDoor()
	{
		Console.WriteLine("In Static CDoor Ctor");
	}
	public static int DoorCount
	{
		get { return m_DoorCount; }
	}
}

public class CStaticExample2
{
	public static int Main(string[] args)
	{
		Console.WriteLine("In Main()");
		CDoor door1;
		door1 = new CDoor();
		return 0;
	}
}
```

פלט התוכנית הנ"ל יהיה:

```
In Main()
In Static CDoor Ctor
In CDoor::Ctor
```

שימו לב: הפונקציה הסטטית לא נקראה לפני Main, אלא לפני יצירת האובייקט הראשון.

10.9. העברת אובייקטים (משתני מחלקות) לפונקציות

כאשר אנו מעבירים אובייקט לפונקציה, מועברת תמיד התייחסות אל האובייקט, ולא עותק שלו. נביט בדוגמא הבאה: אנו יוצרים מחלקה פשוטה השומרת מספר (CNumber), ומאפשרת לשנות ולקבל אותו, וכל אנו יוצרים מחלקה שנייה (CParamExample), המפעילה פונקציות שונות המקבלות כפרמטר אובייקט של המחלקה הראשונה.

```
using System;

public class CNumber
{
	private int m_number;
	public CNumber(int num) { m_number = num; }
	public int number
	{
		get { return m_number; }
		set { m_number = value; }
	}
}

public class CParamsExample
{
	public void f1(CNumber n)
	{
		n.number = 10;
	}
```

```
        public static void f1(ref CNumber n)
        {
                n.number = 10;
        }

        public static void f2(CNumber n)
        {
                n.number = 10;
        }

        public static int Main(string[] args)
        {
                CNumber n1, n2;
                n1 = new CNumber(5);
                n2 = new CNumber(5);
                Console.WriteLine("n1 = {0} and n2 = {1}",
                        n1.number, n2.number);
                f1(ref n1);
                f2(n2);
                Console.WriteLine("n1 = {0} and n2 = {1}",
                        n1.number, n2.number);
                return 0;
        }
}
```

הקוד המודגש הוא המעניין אותנו על מנת לראות את ההבדל בין העברת אובייקט להעברת ערך. כאשר אנו מעבירים מחלקה לפונקציה, התוצאה של שתי הקריאות הנ"ל לפונקציות היא זהה. בשני המקרים, גם כשקראנו ל-f1() וגם כשקראנו ל-f2(), הועברה התייחסות למחלקה ולא עותק שלה.

בהדפסה השנייה על המסך בפונקציה Main(), יודפס כי גם n1 וגם n2 שווים ל-10.

10.10. מבנים והעברת פרמטרים לפונקציות

מלבד מחלקות, C# מכילה גם **מבנים (struct)**.
מבנים מוכרים לנו גם משפות מוקדמות יותר (C/C++) אולם בשפת C# השתנתה משמעותם.

- **C++**: ההבדל בין מחלקות למבנים היה ברמת הגישה בברירת המחדל שלהם. במחלקה רמת הגישה של ברירת המחדל הייתה private ואילו ב-struct רמת הגישה הייתה בברירת המחדל public. הצורך במבנים ובהתנהגות זו שלהם נבע מהרצון של יוצרי השפה לאפשר להריץ קוד שנכתב בשפת C על קומפיילרים של שפת C++.
  מובן זה של struct איננו קיים יותר ב-C#, מכיוון שאנו מגדירים את ההרשאה לפני כל משתנה.
- **C#**: גם בשפת C#, מבנים דומים מבחינה תחבירית למחלקות. כאשר ניצור מבנה נשתמש במילה struct במקום class, וכך נזהה שהישות שאנו יוצרים כרגע היא מבנה חדש ולא מחלקה.
  ההבדל בין מבנה למחלקה ב-C# הוא כלהלן:
  כפי שראינו, כאשר מעבירים מחלקה לפונקציה, מועברת תמיד התייחסות אליה.
  לעומת זאת אם נעביר מבנה לפונקציה, ולא נבקש במפורש להעביר התייחסות אליו, ייוצר עותק של המבנה, והוא זה שיועבר לפונקציה.

הכרות נוספת עם מבנים – תעשה בפרק בהמשך הספר.

11. מחרוזות

11.1. מבוא ותחביר

מחרוזת היא אוסף של תווים סדורים.
בשפת C מימשנו מחרוזות על ידי מערך של תווים. בשפת ++C השפה אפשרה התייחסות למחזורות כאל מערכי תווים, או אפשרה שימוש במחלקה שהיתה חלק ממחלקת הספריות של השפה על מנת לבצע פעולות על מחרוזות.
מחרוזות הן אחד החלקים החשובים של השפה – מכיוון שהמשתמשים האנושיים מבינים אותן ויראו אותן. מחרוזות יוצרות מילים, משפטים והודעות שיכולות להיות מוצגות לבני אדם.
כל אחת מהשפות השונות מודעת לחשיבות של המחרוזות, ולכן הגדירה בשפה תוספות מיוחדות כדי ליעל את העבודה עם מחרוזות.
שפת #C התקדמה צעד נוסף קדימה, ובשפה מוגדר טיפוס מיוחד לצורך אחזקת מחרוזות - string. המחרוזת השמורה ב-string מכילה תווים מסוג UNICODE, כך שהיא תומכת בכל השפות הקיימות בעולם.

דוגמאות להגדרת מחרוזות:

```
string s1 = "Hello";
string s2 = "World";
```

מחרוזות הן משתנים שאינם מוקצים על המחסנית, אלא על הערימה. כאשר אנו יוצרים מחרוזת, נוצר למעשה אובייקט חדש (למרות שאנו לא משתמשים במקרה זה ב-new), ולכן s1, למשל, הוא משתנה המתייחס לאובייקט מסוג string, המכיל את המחרוזת "Hello".

הטיפוס string הוא משתנה המשתמש לקריאה בלבד, ולא ניתן לשנות אותו בזמן ריצה.

דוגמא:

```
public class CStringExample1
{
	public static int Main(string[] args)
	{
		string s1 = "Hello";
		char ch = s1[2];  // Ok
		s1[2] = 'x';		// Error
	}
}
```

אנו יכולים לקבל את תווי המחרוזת, אך איננו יכולים לשנות אותם.

למחלקה string פונקציות רבות שמשמשות למניפולציה על מחרוזות. פונקציות המשנות את המחרוזת יוצרות למעשה מחרוזת חדשה הכוללת את השינוי, ולא משנים את המחרוזת עצמה.

דוגמא: הפונקציה Insert מוסיפה למחרוזת קיימת מחרוזת חדשה, באינדקס שצוין. המחרוזת המקורית למעשה איננה משתנית, אלא מוחזרת מחרוזת חדשה, לאחר התוספת:

```
public class CStringExample1
{
	public static int Main(string[] args)
	{
		string s1 = "Hello";
		string s2 = s1.Insert(s1.Length, " World");
		Console.WriteLine(s1);
		Console.WriteLine(s2);
		return 0;
	}
}
```

אחרי הקריאה לפונקציה Insert, המחרוזת s1 ממשיכה להחזיק בתוכנה המקורי, ואילו המחרוזת s2 תחזיק במחרוזת "Hello World".

ניתן להשתמש גם באופרטור + על מנת לחבר בין מחרוזות.

תוצאת הדוגמא הבאה זהה לתוצאת הדוגמא הקודמת:

```
using System;


public class CStringExample1
{
	public static int Main(string[] args)
	{
		string s1 = "Hello";
		string s2 = s1 + " World";
		Console.WriteLine(s1);
		Console.WriteLine(s2);
		return 0;
	}
}
```

StringBuilder .11.2

המחלקה StringBuilder מאפשרת לנו לעבוד על סוג של מחרוזת אותה ניתן לשנות ללא יצירת מחרוזת חדשה.

על מנת להשתמש במחלקה זו, נצטרך להוסיף גם את System.Text לתוכנית שלנו:

```
using System;
using System.Text;

public class CStringExample1
{
	public static int Main(string[] args)
	{
		string s1 = "Hello";
		StringBuilder sb1 = new StringBuilder(s1);
		Console.WriteLine(sb1);
		sb1[1] = 'Z';
		Console.WriteLine(sb1);
		return 0;
	}
}
```

נקודות אליהן חשוב לשים לב בדוגמא:

- **Namespaces**: על מנת להשתמש ב-StringBuilder עלינו להוסיף את ה-System.Text Namespace (מבוצע בשורה השניה בקוד).
- **יצירת StringBuilder**: העברנו בפונקציה הבונה מחרוזת רגילה, המשמשת כתוכן ההתחלתי של ה-StringBuilder.
- **אינדקסים**: האינדקס של האיבר הראשון במחרוזת הינו 0. כאשר שינינו את התא sb1[1] התא השני במחרוזת הושפע.

נביט בדוגמא נוספת:

```
using System;
using System.Text;

public class CStringExample1
{
	public static int Main(string[] args)
	{
		string s1 = "Hello";
		int i = 10, j = 20;
		StringBuilder sb1 = new StringBuilder(s1);
		Console.WriteLine(sb1);
		sb1.Length = 0;
		sb1.AppendFormat("({0}, {1})", i, j);
		Console.WriteLine(sb1);
		return 0;
	}
}
```

יכולות נוספות של StringBuilder שהכרנו בדוגמא זו:

- **ריקון המחרוזת**: כאשר הגדרנו את האורך של המחרוזת ל-0, רוקנו למעשה את המחרוזת.
- **AppendFormat**: כאשר קראנו לפונקציה AppendFormat, אנו מוסיפים ל-StringBuilder מחרוזת, בפורמט זהה לזה שאנו משתמשים בו בפונקציה WriteLine של ה-Console.

עולה השאלה – מתי צריך להשתמש ב-String ומתי ב-StringBuilder?

התשובה קשורה לשיקולי יעילות. עבור מחרוזות עליהן מבוצעות מעט פעולות שינוי, string הינה המחלקה היעילה יותר. עבור מחרוזות עליהן יבוצעו פעולות רבות, StringBuilder היא הבחירה המומלצת.

כסגנון תכנות – מומלץ להשתמש ב-string וכאשר מתעורר צורך לשפר ביצועים בקטע קוד מסויים, לבדוק את האפשרות לשימוש ב-StringBuilder.

11.3. מחרוזות ותווים מיוחדים

ב-#C, כמו ב-++C/C, ישנם אוסף תווים מיוחדים, שמזוהים על ידי הסימן \.
כאשר אנו רוצים להשתמש בתו \, עלינו לכתוב \\, על מנת להבהיר לשפה שמדובר בסימן, ולא באחד מהתווים המיוחדים.
בפרק 5 במסמך זה ניתן לראות רשימה מלאה של התווים המיוחדים השונים.

לדוגמא:

```
string szFile1 = "c:\\folder\\file.ext";
```

#C מאפשרת לנו צורה מקוצרת של כתיבה, אם אנחנו יודעים שבמחרוזת לא הולכים להופיע תווים מיוחדים. אנו מוסיפים @ לפני המחרוזת, ואומרים בכך לקומפיילר להתעלם מהתווים המיוחדים.
ניתן לכתוב את השורה לעיל גם בצורה הבאה:

```
string szFile1 = @"c:\folder\file.ext";
```

לאחר הקומפילציה של הקוד, כמובן שאין כל הבדל, זהו מאפיין של השפה ודרכה להתמודדות עם תווי בקרה המוכנסים לקוד כקבועים, שני הקודים שהובאו מביאים לאותו קוד IL (שמזכיר אגב, את הקוד הראשון, עם \\).

12. מערכים

12.1. הגדרות ודוגמאות ראשונות

מערך הוא רצף של אובייקטים או משתנים בסיסיים, שניתן לגשת אליהם בעזרת אינדקס.
מערכים ב-C# מוגדרים כאובייקטים.

הגדרת מערך נראית כך:

```
type[] name;
```

ב-net 1.1. המערך ממומש עבור כל סוג של המערכת, ועבור סוגים לא ממומשים נוצרת מחלקה המממשת את המערך עבור הסוג הספציפי אוטומטית, לעומת זאת ב-net 2.0. והלאה התרחשו שינויים לצורך הפיכת ה-CLR לגנרי לצורך הוספת יכולות של גנריות לשפות, והמערך הפך לגנרי ודינאמי, וישנו סוג אחד של מערך עבור כל סוגי הערכים.

מספר האיברים במערך נקבע בעת הקצאת האובייקט, ואינו יכול להשתנות לאחר ההקצאה.

דוגמאות להגדרת מערכים:

```
int[] arr1;
long[] arr2, arr3;
```

בדוגמא זו הגדרנו 3 מערכים.

דוגמא להקצאת מערכים:

```
int[] arr1 = new int[10];
int[] arr2;
arr2 = new int[20];

int[] arr3 = new int[] {1, 2, 3, 4, 5 };
int[] arr4 = { 1, 2, 3, 4, 5 };
```

בדוגמא זו אנו רואים את הדרכים השונות להקצאת זיכרון למערך.

- arr1 מוגדר, ומיד אנו מקצים עבורו זיכרון.
- arr2 מוגדר, ושורה לאחר מכן אנו מקצים עבורו זיכרון.
- arr3 מוגדר, וכאשר אנחנו מקצים עבורו זיכרון איננו מציינים במפורש מה הגודל הרצוי לנו, אלא אנו מציינים את האיברים שברצוננו לשים במערך. בדומה ל-C/C++, C# תקצה מערך בגודל שיתאים בדיוק על מנת להכיל את הערכים המבוקשים, ותשים ערכים אלו בו. arr3 יהיה מערך בגודל חמישה תאים.
- arr4 זהה ל-arr3, מלבד זה שאנו משתמשים בהגדרה מקוצרת הזהה בפעולתה לזו של arr3.

נשים לב שכאשר אנו מאתחלים מערך לגודל מסוים איננו חייבים להשתמש בקבוע. מותר לנו גם לשים משתנה שיכיל את הגודל המבוקש של המערך.

12.2. העברת מערכים לפונקציה

מערכים מועברים לפונקצית על ידי התייחסות (reference) כברירת המחדל, לכן הפונקציות מקבלות למעשה את המערך ולא העתק שלו.

דגשים בנוגע להעברת מערכים לפונקציות בשפת C#:

- ב-C# אין צורך להעביר לפונקציה את גודל המערך, כפי שהיה נהוג ב-C/C++. C# מספקת מספר פתרונות אחרים על מנת לגלות את גודל המערך אותם נראה מיד.
- אם נשנה מערך במהלך פונקציה, ישתנה גם המערך המקורי.

12.3. פונקציה המחזירה מערך

פונקציה יכולה גם להחזיר מערך. מערכים ב-C# מוקצים על הערימה ולא על המחסנית, ולכן ניתן להקצאות מערך בתוך פונקציה, ולאחר מכן להחזיר אותו.

דוגמא

```
using System;

public class CArraysExample1
{
	// The function gets an array and print it
	public static void PrintArray(int[] arr)
	{
		foreach(int iValue in arr)
		{
			Console.Write("{0} ", iValue);
		}
		Console.WriteLine();
	}

	// The function gets an array and add 2 to each element in
	// the array
	public static void ChangeArray(int[] arr)
	{
		for (int i = 0; i < arr.GetLength(0); arr[i++] += 2);
	}

	// The function creates new array and returns it
	public static int[] ReturnArray()
	{
		int[] arr = new int[]{1, 2, 3, 4, 5};
		return arr;
	}

	// Main program
	public static int Main(string[] args)
	{
		int[] arr1 = new int[10]{10, 9, 8, 7, 6, 5, 4, 3, 2, 1};
		PrintArray(arr1);
		ChangeArray(arr1);
		PrintArray(arr1);
		int[] arr2;
		arr2 = ReturnArray();
		PrintArray(arr2);
```

```
            return 0;
        }
}
```

נביט בפונקציות השונות:

- PrintArray() מקבלת מערך, ובעזרת הלולאה foreach עוברת על כל האיברים במערך.
- ChageArray() מקבלת מערך, ובעזרת הפונקציה של האובייקט מערך - GetLength(), עוברת על כל איברי המערך. במידה ואנו עובדים על מערך חד ממדי, הפרמטר המועבר ל-GetLength() הוא 0.
- נראה שוב את פונקציה זו כשנציג מערכים דו ממדיים.
- הפונקציה ReturnArray() מקצה מערך ומחזירה אותו.
- ב-C#, בניגוד ל-C/C++, פונקציות יכולות להחזיר מערכים. המערך ממשיך להתקיים גם אחרי שהפונקציה ReturnArray() הסתיימה.

בפונקציה printArr(), ובכלל, ניתן לבצע על כל מערך לולאת foreach משום שמחלקת מערך (System.Array) יורשת את הממשק IEnumerable, תוכל לרשת את הממשק במחלקות משלך ועל ידי כך לאפשר מבנה בקרה זה ואחרים.

12.4. מערכים רב ממדיים

ב-C# שני סוגים של מערכים רב ממדיים - מערכים מלבניים ומערכי jagged.

מערכים מלבניים הם מערכים בהם כל השורות באותו אורך, ואילו מערכי jagged הם מערכים בהם שורות שונות יכולות להיות באורך שונה.

12.4.1. מערכים מלבניים

הגדרת מערך רב ממדי מתבצעת בעזרת רישום פסיקים בתוך הסוגריים המרובעים, כך שיהיה מקום לרשות את כל הממדים של המערך, למשל:

```
int [,] mat;
int[,,] cude;
```

הקצאת המערכים תיעשה בצורה הבאה:

```
mat = new int[10,5];
cude = new int[3,3,3];
```

גישה לאיברים במערך דו ממדי תיעשה בצורה הבאה:

```
mat[2,3] = 3;
cude[1,2,1] = 4;
```

12.4.2. מערך jagged

מערך jagged הוא למעשה מערך חד ממדי, בו כל איבר הוא מערך בפני עצמו.
ראשית נקצה את המערך, ולאחר מכן נקצה כל איבר בו בנפרד.

דוגמא:

```
using System;

public class CArraysExample2
{

      // Main program
      public static int Main(string[] args)
      {
            int [][]arr;
            arr = new int[4][];      // four rows in the array
            arr[0] = new int[10];    // 10 cells in the first line
            arr[1] = new int[7];     // 7 cells in the second line
            arr[2] = new int[2];     // 2 cells in the third line
            arr[3] = new int[24];    // 24 cells in the last line

            arr[3][2] = 2;           // access an element
            return 0;
      }
}
```

12.5. שיטות ומאפיינים של מערכים

למערכים יש מספר שיטות (Methods) ומאפיינים, בהם אנו יכולים להשתמש.

מאפיין: Rank

תיאור: מאפיין המחזיר את כמות הממדים של המערך.

דוגמא:

```
int[,,] cude;
cude = new int[7,8,9];
Console.WriteLine(cude.Rank);
```

הערך שיודפס יהיה 3.

מאפיין: Length

תיאור: מחזיר את גודל המערך, שהוא מכפלת אורכי כל הממדים שלו.

דוגמא:

```
int[,,] cude;
cude = new int[10, 20, 30];
Console.WriteLine(cude.Length);
```

הערך שיודפס יהיה $10 \cdot 20 \cdot 30 = 6000$.

שיטה: Clear()

תיאור: פונקציה המאפסת תחום ערכים עבור משתנים המחזיקים ערך, ומאתחלת ל-null משתנים המכילים ערך התייחסות.

דוגמא:

```
int[] arr = { 1, 2, 3, 4, 5, 6, 7, 8 };
System.Array.Clear(arr, 2, 3);
for (int i = 0; i < arr.Length; ++i) Console.Write("{0} ", arr[i]);
Console.WriteLine();
```

יודפס: 1 2 0 0 0 6 7 8

שיטה: Clone()

תיאור: פונקציה זו יוצרת העתק של המערך ומחזירה אותו.

דוגמא:

```
int[] arr1 = { 1, 2, 3, 4, 5, 6, 7, 8 };
int[] arr2 = (int[])arr1.Clone();
```

שיטה: Sort()

תיאור: פונקציה זו מבצעת מיון של המערך המתקבל כפרמטר. הפונקציה יכולה לבצע מיון של אובייקטים או מבנים בתנאי שהם ממשים את IComparable Interface (אותו נראה בהמשך המסמך).

דוגמא:

```
int[] arr = { 7, 5, 3, 4, 1, 9, 2, 8 };
System.Array.Sort(arr);
for (int i = 0; i < arr.Length; ++i)
      Console.Write("{0} ", arr[i]);
Console.WriteLine();
```

שיטה: GetLength()

תיאור: פונקציה זו מחזירה את הגודל של מימד מסוים במערך.

דוגמא:

```
int[,] mat;
mat = new int[10,20];
Console.WriteLine("GetLength(0) = {0} and GetLength(1) = {1}",
      mat.GetLength(0), mat.GetLength(1));
```

יודפס:

GetLength(0) = 10 and GetLength(1) = 20

שיטה: IndexOf()

תיאור: הפונקציה מחזיקה את האינדקס של המופע הראשון של ערך מסוים. הפונקציה מקבל ערך כפרמטר, ומחזירה את האינדקס של האיבר, או 1- אם לא נמצא איבר כזה. ניתן להפעיל פונקציה זו רק על מערכים חד ממדיים.

דוגמא:

```
int[] arr = { 4, 56, 23, 2, 43, 23, 12, 83 };
Console.WriteLine(System.Array.IndexOf(arr, 43));
```

שיטה: Binary Search()

תיאור:פונקציה זו מחפשת ערך במערך, לפי אלגוריתם של חיפוש בינארי. על המערך להיות ממוין.

דוגמא:

```
int[] arr = { 1, 34, 45, 56, 61, 78, 88, 89, 2937 };
Console.WriteLine(System.Array.BinarySearch(arr, 61));
```

13. מבנים תחביריים נוספים בשפה

13.1. מבנים

כפי שכבר ראינו, C# מכילה מבנים (struct), בנוסף למחלקות. כאשר מעבירים מחלקה לפונקציה, מועברת תמיד התייחסות אליה ולעומת זאת, אם נעביר מבנה לפונקציה, ולא נבקש במפורש להעביר התייחסות אליו, ייוצר עותק של המבנה, והוא זה שיועבר לפונקציה. מבנה הוא בעל סמנטיקה של ערכים. הוא מוקצה על המחסנית ולא על הערימה. בנוסף – הוא קיים בסקופ של הפונקציה בה הוא מוגדר – כשהפונקציה מסיימת את פעולתה, מפסיק המבנה להתקיים.

דגשים לגבי מבנים:

- **גודל המבנה**: במבנים מומלץ להשתמש בעיקר עבור אובייקטים עם מעט נתונים. מיקרוסופט ממליצים שבאופן אידיאלי גודלו של struct לא יהיה מעבר ל-16 bytes.
- **פונקציות בונות**: לא ניתן ליצור מבנה בעל בנאי ללא פרמטרים. אנו יכולים לממש רק בנאים עם פרמטרים. הקומפיילר יוסיף בכל מקרה בנאי ברירת מחדל בלי פרמטרים, שיאפס את כל שדות המבנה.
- **הורשה**: בהמשך נראה מחלקות הנורשות ממחלקות אחרות. מבנים לעומת מחלקות אינם יכולים לרשת ממבנים או מחלקות אחרים. כמו כן הם אינם יכולים להוות את הבסיס למחלקות.
- **פונקציות הורסות**: לא ניתן לכתוב פונקציה הורסת (destructor) עבור מבנים. לכאורה היה יכול להיות יתרון בשימוש בפונקציה הורסת עם מבנה, מכיוון שידוע בדיוק מתי מבנה נהרס (סוף הפונקציה בה הוא הוגדר) ואז היה ניתן לעשות ניקוי זכרון דטרמיניסטי לחלוטין. עם זאת, שפת C# בחרה שלא לאפשר יצירת פונקציות הורסות עבור מבנים – נסיון לעשות כך יגרור שגיאת קומפילציה.

- **יצירת מבנים**: ניתן ליצור מבנים ללא שימוש באופרטור new. במקרה כזה, נוכל להשתמש במבנה רק אחרי שנאתחל את כל השדות שבו.

```
struct test
{
	public int i;
}

public class CStructExample1
{
	public static int Main(string[] args)
	{
		test t;
		t.i = 3;
		Console.WriteLine(t.i);
		return 0;
	}
}
```

בדוגמא זו, אם נמחק את השורה המודגשת, נקבל שגיאת קומפילציה.

דוגמא לשימוש במבנה:

```
using System;

public struct Point
{
	public int x, y;

	public Point(int px, int py)
	{
		x = px;
		y = py;
	}
}
```

```
class MainClass
{
	public static void Main()
	{
		// Initialize:
		Point myPoint = new Point();
		Point yourPoint = new Point(10,10);

		// Display results:
		Console.Write("My Point:   ");
		Console.WriteLine("x = {0}, y = {1}", myPoint.x,
			myPoint.y);
		Console.Write("Your Point: ");
		Console.WriteLine("x = {0}, y = {1}", yourPoint.x,
			yourPoint.y);
	}
}
```

והתוצאה תהיה:

My Point: x = 0, y = 0

Your Point: x = 10, y = 10

Enumerations .13.2

בעזרת המילה השמורה enum אנו יכולים להגדיר סט של קבועים.

השימוש הבסיסי דומה לשימוש ב-C/C++:

```
using System;

public class EnumTest
{
	enum Days {Sun = 1, Mon, Tue, Wed, Thu, Fri, Sat};

	public static int Main()
	{
		int x = (int) Days.Sun;
		int y = (int) Days.Fri;
		Console.WriteLine("Sun = {0}", x);
		Console.WriteLine("Fri = {0}", y);

		return 0;
	}
}
```

נשים לב רק לעובדה כי ב-C# הקבועים משויכים ל-enum, כלומר עלינו לכתוב Days.Sun על מנת להשתמש באחד מאברי ה-enum, ואיננו יכולים להסתפק רק בכתיבה של הביטוי Sun למטרה זו.

כמו כן, נביט בדוגמא הבאה:

```
using System;

class MainClass
{
      enum colors { BLACK, WHITE, RED, BLUE };

      public static int Main()
      {
            colors i = colors.WHITE;
            Console.WriteLine("{0}", i);
            return 0;
      }
}
```

על המסך תודפס המילה WHITE, ולא הערך המספרי 1.

כברירת מחדל, enum מוחזק בזיכרון כמשתנה מטיפוס של int. ערכו של הקבוע הראשון הוא אפס וערכו של כל קבוע אחריו גדל באחד. ניתן לשנות גם את סוג המשתנה וגם את ערכי הקבועים:

```
enum grades : short { FAIL = 40, PASSED = 55, EXCELLENT = 100 };
```

סוגי הטיפוס המותרים ל-enum הינם: byte, sbyte, short, ushort, int, uint, long, ulong.

מתחת לפני השטח נוצרת מחלקה היורשת את System.Enum ומכילה מספר שדות קבועים, ההמרה למחרוזת נעשית באמצעות שם כל שדה וההמרה מערך בחזרה ל-Enum מתבצעת באמצעות מילון (Hash Table) אשר מתחולל במחלקה.

13.3. קבועים

הגדרת קבועים ב-C# נעשית בעזרת המילה השמורה const.

קבוע הוא ערך שאינו משתנה במהלך כל ריצת התוכנית. כל קבוע שייך למחלקה כלשהי, והגישה לקבוע נעשית בעזרת שם המחלקה. קבוע איננו שייך לאובייקט כלשהו של המחלקה.

דוגמא:

```
using System;

class MathClass
{
	public const double PI = 3.142;
}

class MainClass
{
	public static int Main()
	{
		Console.WriteLine("PI = {0}", MathClass.PI);
		return 0;
	}
}
```

השימושים לקבועים הם לצרכי קריאות הקוד (שמות משמעותיים לערכים) ולצרכי הגדרות כלליות הקשורות למחלקה.

readonly .13.4

readonly זוהי תכונה חדשה ש-C# הוסיפה למשתנים.
משתנה המוגדר כ-readonly ניתן לאתחול בפונקציה הבונה בלבד, ולאחר מכן ניתן רק לקרוא את ערכו.
בתכנות ישנם משתנים חברים רבים המציגים התנהגות זו – אתחול עם יצירת האובייקט ולאחר מכן שמירה על ערך קבועה. התכונה readonly הופכת התנהגות זו לחלק מהשפה, ועוזרת לנו באיתור באגים – במידה וננסה לשנות בטעות את ערכו של המשתנה בהמשך נקבל הודעה שגיאה. משתנה מסוג readonly הוא משתנה השייך לאובייקט, ולא למחלקה.

דוגמא:

```
using System;

class Person
{
	public readonly string Name;

	public Person(string szName)
	{
		Name = szName;
	}
}

class MainClass
{
	public static int Main()
	{
		Person per1 = new Person("Moshe");
		Console.WriteLine("Person name is {0}", per1.Name);

		// The following line won't pass compile:
		// per1.Name = "David";
		return 0;
	}
}
```

Boxing, Unboxing .13.5

פעולת Boxing ב-C# היא הפיכת משתנה ערך למשתנה התייחסות.
הפעולה Unboxing היא הפעולה ההפוכה.

פעולת boxing נעשית על ידי השמת משתנה ערך למשתנה מטיפוס object.

דוגמא:

```
CPoint p1 = new CPoint();
object o = p1;
```

כדי להפוך משתנה התייחסות למשתנה ערך, נשתמש ב-Casting:

דוגמא:

```
CPoint p = (CPoint)o;
```

בפעולות Boxing ו-Unboxing נתקל כאשר נכניס ערך לתוך רשימה (פעולת boxing – מכיוון שהרשימה תשמור התייחסות אל הערך), כאשר נוציא ערך מרשימה (על מנת לגשת לערך עצמו, נצטרך לבצע המרה לטיפוס המתאים), ובמקרים רבים נוספים.

חשוב לשים לב שפעולות אלו קורות – המרה לא מתבצעת באופן אוטומטי, אלא הסביבה מבצעת את פעולת ה-boxing/unboxing. כתוצאה מכך חלק משגיאות המרת האובייקטים יתגלו רק בזמן ריצה. בנוסף לעתים בתור אמצעי לייעל תוכניות, ננסה למזער את כמות הפעמים שפעולות אלו מתרחשות.

13.6. הוראות למהדר

ניתן לתת ב-#C הוראות למהדר על מנת לבצע קומפילציה מותנית.

הפקודות הן דומות לאלו של ++C/C, מבחינת הכתיבה ומבחינת המשמעות, מלבד העובדה שלא precomplier הוא זה המבצע אותן אלא הקומפיילר עצמו.

הפקודות הן #define, #undef, #if, #else, #endif.

בנוסף נוכל להשתמש ב #error וב-#warning על מנת להתריע על בעיות.

דוגמא:

```
#if  ! SIMULATOR
#warning No Simulator defined
#endif
```

14. הורשה

14.1. הורשה בשפת C#

שפת C# תומכת במחלקות היורשות ממחלקות אחרות. מחלקה היורשת ממחלקה אחרת מקבלת את כל התכונות של המחלקה המקורית ומסוגלת להרחיב אותה ולשנות את הפונקציונליות שלה.
בניגוד ל-C++, ב-C# מותרת הורשה ממחלקה אחת בלבד (אין הורשה מרובה), וכן אין סוגי הורשה שונים.

השוואה בין מנגנון ההורשה של שפת C# למנגנון ההורשה של C++:

- ב-C# כל ההורשות הן מסוג public, לעומת C++ שמאפשרת הורשות מצורות שונות.
- כאמור, בשפת C# אין הורשה מרובה ממספר מחלקות.
- כן ניתן לבצע הורשה ממספר interface שונים (ועל כך בהמשך).

מחלקה הנורשת ממחלקה אחרת מקבלת את כל תכונותיה.
הצורה הכללית של הגדרת מחלקה הנגזרת ממחלקה אחרת היא:

```
Modifiers class derived_class_name : base_class_name
{
	// Body of class goes here
}
```

כאשר:

- derived_class_name זו שם המחלקה החדשה שאנחנו יוצרים.
- base_class_name זוהי המחלקה ממנה אנו יורשים.

14.2. דוגמא ראשונה

```
using System;

class CPoint
{
	public int x, y;
}

class CPoint3D : CPoint
{
	public int z;
}

class MainClass
{
	public static int Main()
	{
		CPoint3D myPoint = new CPoint3D();
		myPoint.x = 3;
		myPoint.z = 10;
		return 0;
	}
}
```

נקודות:

- הגדרנו מחלקה בשם CPoint, המכילה את המשתנים x, y.
- הגדרנו מחלקה שנייה, בשם CPoint3D, הנגזרת מהמחלקה CPoint, ומכילה משתנה בשם z.
- המחלקה CPoint3D מכילה, בנוסף למשתנה z, גם את המשתנים x, y שירשה מ- CPoint.
- הפונקציה ()Main ניגשת אל משתנים אלו.

14.3. גישה אל שדות private

מחלקה היורשת ממחלקה אחרת, יכולה לגשת אל שדות ה-public שלה, אך לא אל שדות ה-private.

נביט בדוגמא הבאה:

```
class x
{
	private int i;
	public int GetI() { return i; }
}

class y : x
{
	public void f()
	{
		GetI();

		//i = 3;
	}
}
```

המחלקה y ירשה את המשתנה i ואת הפונקציה GetI() מהמחלקה x.

עם זאת, היא יכולה לפנות רק אל הפונקציה GetI().

פנייה אל המשתנה i תגרום לשגיאת קומפילציה, מכיוון שהוא משתנה פרטי של x.

14.4. אתחול מחלקת הבסיס

כאשר אנו יוצרים אובייקט חדש, שנגזר מאובייקט אחר, נרצה לקרוא ראשית לפונקציה הבונה של האובייקט הראשון. על מנת לעשות זאת, נשתמש במילה השמורה base, המתייחסת אל המחלקה ממנה נגזרה המחלקה הנוכחית (בניגוד ל-C++, ב-C# יש בהכרח רק אחת כזאת).
כמו כן נשתמש במילה השמורה base על מנת לגשת אל הפונקציות של המחלקה ממנה גזרנו.

כדי לומר כי פונקציה במחלקה הנגזרת מחליפה פונקציה במחלקה הקוראת, נשתמש באופרטור new. נביט בדוגמא:

```
using System;
class Rectangle
{
      private int m_x, m_y;

      public Rectangle(int x, int y)
      {
            m_x = x;
            m_y = y;
      }
      public int GetX() { return m_x; }
      public void SetX(int new_x) { m_x = new_x; }
      public int GetY() { return m_y; }
      public void SetY(int new_y) { m_y = new_y; }
      public int GetArea() { return m_x * m_y; }
}


class Square : Rectangle
{
      public Square(int len) : base(len, len)
      { }

      new public void SetX(int new_x)
      {
            base.SetX(new_x);
            base.SetY(new_x);
```

```
	}

	new public void SetY(int new_y)
	{
		SetX(new_y);
	}
}

class MainClass
{
	public static int Main()
	{
		Square r1 = new Square(4);
		Console.WriteLine(r1.GetArea());
		r1.SetY(5);
		Console.WriteLine(r1.GetArea());
		return 0;
	}
}
```

14.5. הרשאה protected

משתנים או פונקציות המקבלים הרשאה מסוג protected, בדומה לשפת ++C, נגישים במחלקה בה הם הוגדרו, במחלקות הנגזרות ממנה, אך לא מחוץ אליהם.
אנחנו משתמשים בהרשאה זו כמו שאנחנו משתמשים ב-private או public – פשוט מגדירים אותה לפני האלמנט שאנחנו רוצים לתת לו הרשאה זו.

לדוגמא:

```
protected int i;
```

Sealed Class .14.6

שפת #C מאפשרת לנו ליצור מחלקות שלא יהיה ניתן לגזור מהן מחלקות חדשות.

מחלקה שלא ניתן לגזור ממנה נקראת **מחלקה חתומה**.

הגדרת מחלקה חתומה נעשית בעזרת המילה השמורה **sealed**.

דוגמא:

```
sealed class MyClass
{

}


/*
 * The following line will result in compile-time error:
class MyClass2 : MyClass { }

*/
```

15. פולימורפיזם

15.1. הצגת הצורך

רב צורתיות היא אחת התכונות החזקות של שפת תכנות מונחית עצמים.
רב הצורתיות ממומשת בעזרת פונקציות וירטואליות.
הרעיון: נגדיר את מחלקת הבסיס, ונגדיר חלק מהפונקציות בה כ-virtual.
מחלקות הנגזרות ממחלקה זו, יוכלו להציע מימוש משלהן לשדות שהוגדרו virtual.
כעת, אם ניצור פונקציה המקבלת אובייקטים מסוג מחלקת הבסיס, אולם נעביר לה אובייקטים מסוג המחלקות הנגזרות, תיקרא הפונקציה הנכונה של המחלקות הנגזרות, ולא הפונקציה של מחלקת הבסיס.
אנו משתמשים במילה השמורה override כאשר אנו מממשים את הפונקציה הוירטואלית במחלקה הנגזרת.

דוגמא:

```
using System;

class Rectangle
{
	private int m_x, m_y;

	public Rectangle(int x, int y)
	{
		m_x = x;
		m_y = y;
	}

	public int GetX() { return m_x; }
	public virtual void SetX(int new_x) { m_x = new_x; }
	public int GetY() { return m_y; }
	public virtual void SetY(int new_y) { m_y = new_y; }

	public int GetArea() { return m_x * m_y; }
}
```

```
class Square : Rectangle
{
	public Square(int len) : base(len, len)
	{ }

	public override void SetX(int new_x)
	{
		base.SetX(new_x);
		base.SetY(new_x);
	}

	public override void SetY(int new_y)
	{
		SetX(new_y);
	}
}

class MainClass
{
	public static int Main()
	{
		Square r1 = new Square(4);
		Console.WriteLine(r1.GetArea());
		r1.SetX(5);
		Console.WriteLine(r1.GetArea());
		SetLengthTo7(r1);
		Console.WriteLine(r1.GetArea());
		return 0;
	}

	public static void SetLengthTo7(Rectangle r)
	{
		r.SetX(7);
	}
}
```

ניתן לאכוף גם properties. נראה כעת דוגמא לכך.

נגדיר ווקטור דו ממדי, ולאחר מכן נהפוך אותו לתלת ממדי:

המשתנים בדוגמא הם גלובליים, למרות שהדבר איננו מומלץ, וזו על מנת לפשט את הדוגמא.

```
using System;

class Vector
{
	public int x, y;

	public virtual double length
	{
		get
		{
			return System.Math.Sqrt(x*x + y*y);
		}
	}
}

class Vector3D : Vector
{
	public int z;
	public override double length
	{
		get
		{
			return System.Math.Sqrt((base.length*base.length) +
z*z);
		}
	}
}

class MainClass
{
	public static int Main()
	{
		Vector v1 = new Vector();
		v1.x = 10;
		v1.y = 5;
```

```
            Console.WriteLine("Vector Length: {0}", v1.length);

            Vector3D v2 = new Vector3D();
            v2.x = 0;
            v2.y = 1;
            v2.z = 1;
            Console.WriteLine("Vector Length: {0}", v2.length);
            return 0;
      }
}
```

15.2. מחלקות אבסטרקטיות ופונקציות טהורות

לעיתים נרצה ליצור מחלקות, שישמשו לצורך גזירה בלבד, ולא ליצירת אובייקטים מהטיפוס שלהן, מחלקות אלו נקראות מחלקות אבסטרקטיות.

בהגדרת הפונקציה, נשתמש במילה השמורה abstract על מנת לציין שהמחלקה היא אבסטרקטית ולא ניתן להגדיר אובייקטים ממנה.

דוגמא:

```
using System;

abstract class Food
{
      public virtual void eat() { }
}

class Banana : Food
{
}

class MainClass
{
      public static int Main()
      {
            // ERROR:
            // Food f = new Food();
```

```
            // Fine:
            Banana b = new Banana();

            return 0;
        }
}
```

תחת מחלקות אבסטרקטיות, ניתן להגדיר פונקציות אבסטרקטיות. ההגדרה מתבצעת בעזרת המילה השמורה abstract לפני הגדרת הפונקציה. כאשר אנו מגדירים פונקציה כמופשטת, אנו מגדירים את שמה ואת ההצהרה עליה, אולם איננו מממשים אותה. מחלקות הנגזרות ממחלקה אבסטרקטית, חייבות להשתמש במילה override כדי לאכוף את הפונקציות האבסטרקטיות.

כמו כן, אנו יכולים להגדיר גם properties כאבסטרקטיות.

דוגמא:

```
using System;
abstract class MyBaseC   // Abstract class
{
        protected int x = 100;
        protected int y = 150;
        public abstract void MyMethod();   // Abstract method

        public abstract int GetX   // Abstract property
        {
                get;
        }

        public abstract int GetY   // Abstract property
        {
                get;
        }
}

class MyDerivedC: MyBaseC
{
        public override void MyMethod()
```

```
        {
                x++;
                y++;
        }

        public override int GetX    // overriding property
        {
                get
                {
                        return x+10;
                }
        }

        public override int GetY    // overriding property
        {
                get
                {
                        return y+10;
                }
        }

        public static void Main()
        {
                MyDerivedC mC = new MyDerivedC();
                mC.MyMethod();
                Console.WriteLine("x = {0}, y = {1}", mC.GetX, mC.GetY);
        }
}
```

15.3. Casting

כפי שראינו, ניתן להשתמש במשתנה התייחסות של מחלקת בסיס, על מנת להצביע אל אובייקטים של מחלקות הנגזרות ממנה.
אם נרצה להתייחס למשתנים ופונקציות של המחלקה הנגזרת, נוכל לבצע המרה בחזרה אל הטיפוס המקורי של המחלקה, בעזרת casting.
לאחר שנעשה המרה, נוכל שוב לגשת אל המשתנים והפונקציות של המחלקה הנגזרת.
אם האובייקט בפועל הוא לא מהטיפוס של המחלקה אליה ביצענו המרה, ישלח exception.

ניתן לבדוק אם משתנה הוא מסוג מסוים או אם ירש אותו על ידי שימוש במילה השמורה is:

```
MyDerivedC c = new MyDerivedC();
if (!c is MyBaseC)
      Console.WriteLine("You will never see this message");
```

15.4. ביצוע override לפונקציות של object

כל המחלקות ב-C# נגזרות אוטומטית ממחלקת השורש object.
למחלקה זו מספר פונקציות public, שניתן להגדירן מחדש במחלקה הנגזרת.

הפונקציות הן:

```
public virtual string ToString();
public virtual int GetHashCode();
public virtual bool Equals(object o);
public virtual Type GetType();
```

- **ToString** – הפונקציה מחזירה מחרוזת המתארת את האובייקט.
- **GetHashCode** – מחזירה מספר המייצג את האובייקט הספציפי. המספר צריך להיות ייחודי כך שלא יהיו שני אובייקטים בעלי אותו מספר אם הם שונים. הפונקציה משמשת להכנסת האובייקטים למבנה מסוג Hash Table. אם מגדירים מחדש את הפונקציה Equals, יש להגדיר גם פונקציה זו.

- **Equals** – פונקציה זו מבצעת השוואה בין שני אובייקטים. כברירת המחדל, הפונקציה מבצעת השוואה של ערכי ההתייחסות בלבד, כלומר מחזירה true אם המשתנים מצביעים אל אותו אובייקט.

דוגמא:

```
using System;

class MyClass
{ }

class MainClass
{
	public static int Main()
	{
		MyClass c = new MyClass();
		MyClass c2 = c;
		MyClass c3 = new MyClass();
		Console.WriteLine(c.Equals(c2));
		Console.WriteLine(c2.Equals(c2));
		Console.WriteLine(c2.Equals(c3));
		return 0;
	}
}
```

- **GetType** – הפונקציה מחזירה משתנה מסוג Type המכיל אינפורמציה לגבי המחלקה של האובייקט.

<u>פונקציות סטטיות של המחלקה object</u>

- **object.RefrenceEqual** – פונקציה המשווה בין ערכי ההתייחסות של שני אובייקטים.
- **object.Equals** – פונקציה המקבלת שני אובייקטים, ומשווה ביניהם.

16. Interface

16.1. Namespace

בתוכנית גדולה, יתכנו התנגשויות בין שמות מחלקות שונות שאנשים שונים בצוות מפתחים.
בעזרת namespace ניתן לפתור את הבעיה הזו.
נגדיר "מרחב שמות", שיעטוף את המחלקות, ויהיה חלק מהשם המלא של המחלקה.
דוגמא:

```
using System;

namespace mySpace
{
	class MyClass
	{
	}
}

class MainClass
{
	public static int Main()
	{
		mySpace.MyClass myObj = new mySpace.MyClass();

		return 0;
	}
}
```

אם נרצה לקצר את צורת הכתיבה, נשתמש במילה using:

```
using System;
using mySpace;

namespace mySpace
{
	class MyClass
```

```
        {
        }
}

class MainClass
{
        public static int Main()
        {
                MyClass myObj = new MyClass();

                return 0;
        }
}
```

המילה using אומרת למהדר לחפש שמות של מחלקות/מבנים בתוך ה-namespace שצוין, כשהוא בא לקמפל את הקוד. אם ישנם כמה namespace עם מחלקה באותו שם, עדיין לא תתעורר בעיה, מכיוון שיהיה ניתן לפנות למחלקות השונות על ידי השם המלא שלהן.
ניתן להגדיר namespace בתוך namespace. במקרה כזה, הגישה למחלקה פנימית תהיה על ידי רישום ה-namespace החיצוני והפנימי, ולאחריהם שם המחלקה, כשהם מופרדים בנקודות ביניהם.
ניתן גם לתת שם נרדף לשמות. נרצה לעשות זאת במקרה שאנו רוצים לקצר את שמו של namespace שנמצא מקונן עמוק בתוך namespace אחרים.
שוב, נשתמש במילה using, על מנת לעשות זאת:

```
using MyAlias = MyCompany.Proj.Nested;  // define an alias to
represent a namespace

namespace MyCompany.Proj
{
        public class MyClass
        {
                public static void DoNothing()
                {
                }
        }

        namespace Nested    // a nested namespace
        {
```

```
		public class ClassInNestedNameSpace
		{
			public static void SayHello()
			{
				System.Console.WriteLine("Hello");
			}
		}
	}

}

public class UnNestedClass
{
	public static void Main()
	{
		MyAlias.ClassInNestedNameSpace.SayHello();	// using
alias
	}
}
```

interface .16.2

interface היא למעשה מחלקה אבסטרקטית, שבה לא מוגדרים משתנים, וכל הפונקציות בה הן אבסטרקטיות.

מטרת ה-interface הינה להגדיר התנהגות מסוימת, והוא משמש בסיס למחלקות המממשות התנהגות זו.

ההבדל בין interface למחלקה אבסטרקטית שכל הפונקציות בה אבסטרקטיות, הוא שמחלקה יכולה להיגזר ממספר interface, אולם יכולה להיגזר ממחלקה בודדת בלבד. הסיבה לכך היא שישנם שני סוגי הורשות: הורשה של התנהגות והורשה של מימוש. שפת C++ תומכת בהורשה מרובה של מימוש, לעומתה שפת C# תומכת בירושה יחידה של מימוש, ובירושה מרובה של התנהגות.

הגדרת interface נעשית על ידי המילה השמורה interface. דוגמא:

```
interface MyInterface
{
	void Function1(int iValue);
	int Function2();
}
```

אנו מגדירים את הפונקציות בתוך ה-interface. כל הפונקציות הן public וכולן מוגדרות כ-virtual באופן אוטומטי, ולכן אין צורך לרשות דבר פרט לחתימה של הפונקציה – שמה, הפרמטרים שלה וערכים מוחזרים.

Interface יכול להיות public או internal.

internal משמעותו שניתן יהיה להשתמש ב-interface רק באותו assembly ואילו public משמעותו שניתן להשתמש בו בכל assembly.

דוגמא ל-Interface:

```
using System;

interface Aminal
{
	void MakeSound();
}

class Dog : Aminal
{
	public void MakeSound()
	{
		Console.WriteLine("Woof");
	}
}

class Cat : Aminal
{
	public void MakeSound()
	{
		Console.WriteLine("Myaow");
	}
}

class Cow : Aminal
{
	public void MakeSound()
	{
		Console.WriteLine("Muuuu");
	}
}

public class MyClass
{
	public static void Main()
	{
		Dog Dogi = new Dog();
		Cat Shmil = new Cat();
		Cow David = new Cow();
```

```
            printMessage(Dogi);
            printMessage(Shmil);
            printMessage(David);
        }

        static void printMessage(Aminal aminal)
        {
            aminal.MakeSound();
        }
}
```

16.3. IEnumerable, IEnumerator

נושא ה-interfaces הוא נושא חשוב ב-#C. אנו מסוגלים בעזרתם להצהיר על מחלקות שלנו כמחלקות המממשות התנהגות מסוימת, ולממש אותה.
הדוגמה הראשונה שנראה היא מימוש של ה-interfaces הבאים: IEnumerable, IEnumerator. Interfaces אלו מאפשרים לנו ליצור אוסף של אובייקטים, שניתן לבצע עליהם איטרציה באמצעות משפט הבקרה foreach.

כדי להשתמש ב-interface אלו אנו צריכים להצהיר את ההצהרה הבאה לראש התוכנית:

```
using System.Collections;
```

:IEnumerator

```
public interface IEnumerator
{
	bool MoveNext();
	object Current { get; }
	void Reset();
}
```

Public Properties

Current	Gets the current element in the collection.

Public Methods

MoveNext	Advances the enumerator to the next element of the collection.
Reset	Sets the enumerator to its initial position, which is before the first element in the collection.

:IEnumerable

```
public interface IEnumerable
{
	IEnumerator GetEnumerator();
}
```

Public Methods

GetEnumerator	Returns an enumerator that can iterate through a collection.

נביט בדוגמא הממשת את ה-interfaces:

```
using System;
using System.Collections;

class colors : IEnumerator, IEnumerable
{
      private string[] cols;
      private int iCurrent;

      public colors()
      {
            cols = new string[]  { "Red", "Blue", "Green", "White" };
            iCurrent = -1;
      }

      public IEnumerator GetEnumerator()
      {
            return (IEnumerator)this;
      }

      public bool MoveNext()
      {
            if (iCurrent < cols.Length - 1)
            {
                  ++iCurrent;
                  return true;
            }

            return false;
      }

      public object Current
      {
            get
            {
                  return cols[iCurrent];
            }
      }
```

```
	public void Reset()
	{
		iCurrent = -1;
	}
}

public class MyClass
{
	public static void Main()
	{
		colors c = new colors();
		foreach (string item in c)
		{
			Console.WriteLine(item);
		}
	}
}
```

IComparable .16.4

Interface חשוב נוסף הוא IComparable.

כל מחלקה המממשת אותו יכולה להיות ממוינת.

IComparable:

```
interface IComparable
{
	int CompareTo(object o);
}
```

Public Methods

CompareTo	Compares the current instance with another object of the same type.

הפונקציה שה-interface מגדיר היא פונקציה המבצעת השוואה בין האובייקט הנוכחי לאובייקט המתקבל בפרמטר, ומחזירה ערך שלם שיכול להיות חיובי, שלילי או אפס.
אם הערך המוחזר קטן מאפס, אזי האובייקט הנוכחי קטן מהפרמטר. אם הערך גדול מאפס, הפרמטר גדול מהאובייקט הנוכחי, ואם הערך המוחזר הוא אפס, אזי האובייקט הנוכחי שווה לפרמטר.

אם נכתוב מחלקה המממשת interface זה, נוכל למיין אובייקט מהסוג שלה. למשל, אם ניצור מערך של אובייקטים מאותו הסוג, נוכל להשתמש בפונקציה Sort ששייכת למערכים, על מנת למיין את המערך.

16.5. Interface הנגזר ממספר Interfaces

ניתן להגדיר interface הנגזר מכמה interfaces אחרים. במקרה כזה, מחלקות שיממשו interface זה יצטרכו לממש את כל הפונקציות שהוגדרו בכל אחד מה-interface שהוא הוגדר בהם.
ב-++C, אם במחלקות שונות מהן אנו גוזרים מחלקה חדשה יש פונקציות בעלות אותו שם, מתרחשת שגיאת קומפילציה. ב-#C, לעומת זאת, אם נגזור interface חדש ממספר interfaces לא תהיה שגיאת קומפילציה.

נביט בדוגמא לפתרון של #C לבעיה. נגדיר שני interface בעלי פונקציה בעלת אותה חתימה, ניצור מחלקה שתיגזר משניהם, ונראה כיצד אנו מטפלים בהתנגשות השמות.

```
interface interface1
{
	void myFunction(int i);
}

interface interface2
```

```
{
	void myFunction(int i);
}

class dInterface : interface1, interface2
{
	void interface1.myFunction(int i)
	{
		// Function body goes here
	}
	void interface2.myFunction(int i)
	{
		// Function body goes here
	}
}
```

16.6. שימוש בממשק IDisposable

כפי שראינו, כאשר אנו כותבים פונקציה הורסת, היא לא בהכרח נקראת ברגע שאין כבר צורך באובייקט, והיא יכולה להיקרא גם מאוחר יותר..

לעיתים ישנו צורך בשחרור של משאבים מיד כשאין בהם צורך.

על מנת לבצע זאת נשתמש בIDisposable interface.

ל-interface זה פונקציה אחת – Dispose(), המשמשת לשחרור משאבים מייד, עוד לפני שהאובייקט משתחרר.

אם קיים אובייקט הממומש ממחלקה הנגזרת מ-interface זה, ברגע שהתוכנית המשתמשת באובייקט זה רואה שהיא איננה צריכה אותו עוד, הוא קוראת ל-Dispose().

הפונקציה Dispose() איננה מונעת את פעולת הפונקציה ההורסת, ולכן ייתכן שתתבצע פעולה מיותרת.

כדי למנוע מצב כזה, ניתן להפעיל בתוך הפונקציה Dispose() את הפונקציה GC.SuppressFinalize(this). פונקציה זו, השייכת ל-Garbage Collector, גורמת לכך שהפונקציה ההורסת לא תפעל.

פונקציה נוספת של Garbage Collector היא GC.Collect(). פונקציה זו מכריחה את Garbage Collector לשחרר את כל האובייקטים שאין להם התייחסות מיד.

ניתן להשתמש במשפט הבקרה using על מנת שהפונקציה Dispose תיקרא באופן אוטומטי על ידי הקומפיילר בתום בלוק טווח ההכרה של האובייקט.

השימוש בusing נראה כך:

```
using (class1 var1 = new class1(), class2 var2 = new class2(), ... )
{

}
```

בתוך הסוגריים של using ניתן להקצות אובייקטים המממשים את interface IDispose, וכאשר יוצאים מבלוק ההכרה של using הקומפיילר קורא לפונקציה ()Dispose באופן אוטומטי.

היתרון בצורה זו שלא התוכנית צריכה לקרוא ל-Dispose אלא הקומפיילר וכך לא ניתן לשכוח את הקריאה.

17. מגנון בדיקת שגיאות - Exceptions

17.1. מבוא

כל תוכניות מורכבת צריכה מנגנון בדיקת שגיאות מסוים – פונקציות שונות (כגון פונקציות המקצות זיכרון, או פונקציות המנסות לבצע פעולות שונות העלולות להיכשל) צריכות דרך לדווח לקורא להם האם הקריאה לפונקציה הצליחה או נכשלה. כמו כן צריך מנגנון שיתמוך בניתוח השגיאה ותגובה בהתאם.
בשפת C השתמשנו בערכים המוחזרים מהפונקציות כדרך לבדוק את תקינות פעולת הפונקציה.
מנגנון זה הינו מוגבל בתחומים שונים – הוא מסבך את הקוד, (קשה להבחין בין הקוד לבין בדיקת התקלות), במקרים מסוימים, בעיקר עבור הקצאות זיכרון דינאמיות מורכבות, מסובך לשחרר את הזיכרון שהוקצה. כמו כן, לא כל פונקציה יכולה לבחור ערך שייעוד לשגיאה. ייתכנו פונקציות שכל ערך מוחזר יכול להיות גם ערך לגיטימי.
לפיכך ++C ובעקבותיה גם #C תומכים במנגנון exceptions.

הרעיון הראשוני, בדומה ל-++C, הוא הפרדה בין קטע הקוד לקטע בדיקת השגיאות.
את קטע הקוד שאנו כותבים אנו שמים בתוך בלוק try.
במידה ומתרחשת שגיאה, השגיאה לא תוחזר מהפונקציה כערך מוחזר על ידי המילה השמורה return אלא על ידי מנגנון ה-exceptions בעזרת המילה throw.
Exception היא שגיאה אקטיבית – אם לא נטפל בה, התוכנית תפסיק את פעולתה.

דוגמא:

```
using System;

class MainClass
{
       static double Div(double fNumber1, double fNumber2)
       {
              if (fNumber2 == 0)
                     throw new Exception("Divide by zero error.");
              return fNumber1 / fNumber2;
       }
```

```
        static int Main()
        {
                try
                {
                        Console.WriteLine(Div(10, 2));
                        Console.WriteLine(Div(4, 0));
                        Console.WriteLine(Div(8, 2));
                }
                catch (Exception e)
                {
                        Console.WriteLine(e.Message);
                }

                return 0;
        }
}
```

- הפונקציה Div מחלקת שני מספרים ומחזירה את התוצאה. במידה והתחרש חילוק ב-0 היא שולחת Exception.
- ב-Main אנו קוראים שלוש פעמים לפונקציה Div.
- בפעם השניה נשלח Exception, ואנו קופצים אל בלוק ה-cacth שמטרתו לטפל בבעיה, וכלל לא מגיעים אל הקריאה השלישית.
- ב-++C יכולנו לשלוח Exception מכל סוג. ב-#C אנו יכולים לשלוח רק אובייקטים מסוג Exception או שנגזרים ממחלקה זו. אנו יוצרים אובייקט חדש מסוג Exception ושולחים אותו.
- שליחת exception נעשית בעזרת המילה השמורה **throw**.

על מנת לבדוק ולתפוס שגיאות, אנו משתמשים במבנה הבא:

```
        try
        {
        }
        catch (Exception e)
        {
        }
```

כאשר קוראת בפונקציה כלשהי exception, המחסנית שלה משתחררת, והבקרה מועברת אל בלוק

ה-catch שקרא לה. אם אין בלוק כזה, אנו יוצאים גם מהפונקציה הקוראת כלפי מעלה במחסנית, עד שאנו מוצאים קוד המטפל בשגיאה.

מאפיינים של המחלקה Exception:

- Message – מחרוזת המכילה את תיאור הסיבה שגרמה ל-Exception.
- Source – מכיל את שם האובייקט שגרם לשגיאה.
- StackTrace – מכיל מחרוזת של שרשרת הקריאות שהאובייקט Exception עבר עד שהגיע לבלוק ה-catch.

אם נרצה, נוכל ליצור מחלקות שיגזרו מ-Exception, ולתפוס אותם במקום אובייקט מסוג Exception.
בדומה ל-C++, אנו מסוגלים לשים מספר בלוקים של catch אחד אחרי השני, לתפיסת שגיאות מסוגים שונים.

אם קיבלנו שגיאה בבלוק catch, ונרצה לשלוח אותה הלאה, נוכל לעשות זאת באחת משני הדרכים הבאות:

במקרה שהאובייקט הוא מסוג Exception, נכתוב:

```
try
{


}
catch (Exception e)
{
        throw e;
}
```

ניתן גם לתפוס את כל סוגי ה-Exception על ידי רישום catch לא סוגריים. פעולה זו תהיה זהה לרישום catch(…) ב-C++.

דוגמא:

```
try
{

}
catch
{
        throw;
}
```

לעיתים נרצה לבצע פעולות כלשהן, גם אם קרא exception.

במקרה כזה, נוסיף לאחר ה-catch בלוק finally. בלוק זה יתבצע תמיד, בין אם קרא exception או לא, ואפילו אם הייתה פעולת return בתוך בלוק ה-try.

דוגמא:

```
try
{
        Console.WriteLine(Div(10, 2));
        //Console.WriteLine(Div(4, 0));
        Console.WriteLine(Div(8, 2));
        return 0;
}
catch (Exception e)
{
        Console.WriteLine(e.Message);
}
finally
{
        Console.WriteLine("The End...");
}
```

ApplicationException .17.2

ישנן שתי מחלקות בסיס עבור שגיאות ב-Net., האחת היא Exception והשנייה היא ApplicationException, הראשונה נועדה להעלאת שגיאות ממחלקות ה-CLR, דוגמא עבורה היא שגיאה בקריאה מקובץ שלא קיים או ניסיון תקשורת לאתר שאינו קיים, והשנייה נועדה להעלאת שגיאות בתכנית של המשתמש, ולא של סביבת הריצה.

ישנן גם ApplicationException, לדוגמא ExecutionEngineException, StackOverFlowException, וכו'. לא נהוג לתפוס ApplicationException בשל אופיין וחוסר היכולת להתמודד עימן, ואף יותר חמור מכך אין זה נהוג לזרוק שגיאות מסוג זה.

17.3. בניית Exception משלך

ניתן וצריך ליצור לעיתים שגיאות משלך בעלות מידע ייחודי לסוגן שישמר בהן. כדי לעשות זאת, נגזור מחלקה מ-Exception (או ממחלקה הנגזרת ממנה).

להלן דוגמא למחלקת שגיאה:

```
using System;
class MyException : Exception
{
        public MyException(string str)
        {
                Console.WriteLine ("User Defined Exception");
        }
}

class MyClient
{
        public static void Main()
        {
                try
                {
                        throw new MyException ("dugma");
                }
                catch(Exception e)
                {
                        Console.WriteLine(
```

```
                        "Exception caught here" + e.ToString () );
            }
      Console.WriteLine("Last Statement");
      }
}
```

18. Operators Overloading

18.1. חפיפת אופרטורים בינאריים

חפיפת אופרטורים הינה האפשרות להגדיר מחדש את הפעולות הבסיסיות על משתנים, כגון +, -וכו', כך שנוכל להפעיל אותם על התוכנית שלנו.
למשל, נניח שנכתוב מחלקה המייצגת מספרים קומפלקסים, שנקרא לה Complex.
נרצה לדאוג שנוכל לכתוב את התוכנית הבאה:

```
class MainClass
{
	static int Main()
	{
		Complex num1 = new Complex(4, 0);
		Complex num2 = new Complex(5, 7);
		Complex num3 = num1 + num2;
		Console.WriteLine("({0}, {1})", num3.r, num3.i);
		return 0;
	}
}
```

כאשר חיברנו בין num1 ל-num2, נרצה שתתרחש פעולת החיבור ההגיונית של מספרים קומפלקסים.

נביט במימוש אפשרי למחלקה Complex:

```
class Complex
{
	public double r, i;
	public double length
	{
		get
		{
			return System.Math.Sqrt(r*r+i*i);
		}
	}
	public Complex(double dR, double dI)
```

```
        {
                r = dR;
                i = dI;
        }

        public static Complex operator +(Complex num1, Complex num2)
        {
                Complex ret = new Complex(num1.r + num2.r, num1.i +
num2.i);
                return ret;
        }
}
```

בדוגמא זו חפפנו את האופרטור +.

חפיפת אופרטורים נעשית על ידי פונקציות סטטיות בלבד. הפונקציה הסטטית שכתבנו מקבלת שני מספרים קומפלקסיים, מחברת אותם ומחזירה את התוצאה.

האופרטור + הינו אופרטור בינארי, המקבל שני אובייקטים, כמוהו האופרטורים * ,- וכו'.
לעומתם יש גם אופרטורים אונריים, כגון ++, המקבלים רק אובייקט אחד, עליו הם פועלים.
לדוגמא, נוסיף הגדרת אופרטור ++ למחלקה, שיגדיל את החלק הממשי של המספר ב1:

```
	public static Complex operator ++(Complex num)
	{
		num.r++;
		return num;
	}
```

ישנם אופרטורים הבאים בזוגות: אם נרצה ב-C# לחפוף את האופרטור ==, נהיה חייבים לחפוף גם את האופרטור =!.
כמו כן, הגדרת > מחייבת גם את הגדרת <, ואילו הגדרה => מחייבת גם את הגדרת =<.
עבור האופרטור == נצטרך לחפוף גם את האופרטור Equals וגם את GetHashCode.

דוגמא מסכמת:

```
class Complex
{
	public double r, i;
	public double length
	{
		get
		{
			return System.Math.Sqrt(r*r+i*i);
		}
	}
	public Complex(double dR, double dI)
	{
		r = dR;
		i = dI;
	}

	public static Complex operator +(Complex num1, Complex num2)
	{
		Complex ret =
			new Complex(num1.r + num2.r, num1.i + num2.i);
		return ret;
	}
```

```
        public static Complex operator ++(Complex num)
        {
            num.r++;
            return num;
        }

        public override bool Equals(object o)
        {
            if (r == ((Complex)o).r && i == ((Complex)o).i)
        return true;
            return false;
        }

        public static bool operator==(Complex num1, Complex num2)
        {
            if (num1.r == num2.r && num1.i == num2.i) return true;
            return false;
        }
        public static bool operator!=(Complex num1, Complex num2)
        {
            if (num1.r != num2.r || num1.i != num2.i) return true;
            return false;
        }
        public override int GetHashCode()
        {
            return this.GetHashCode();
        }
}
```

18.2. Indexer

ב-C# לא ניתן לחפוף את האופרטור [].

על מנת לחפוף אופרטור זה נשתמש ב-Indexer. הצורה הדקדוקית היא זו:

```
public <return type> this[<input> index]
{
    get
    {
        return  m_value[index];
    }
    set
    {
        m_value[index] = value;
    }
}
```

שתי אפשרויות נוספות שעומדות לרשותנו היא הפיכת הערך לכתיבה בלבד או לקריאה בלבד ע"י השמטת קוד {}set או קוד {}get. לדוגמא, אם לא נכתוב בלוק {}set אז לא נוכל להציב ערכים בתכונה ולכן היא תהיה לקריאה בלבד.

אפשרות נוספת היא העברת מספר מפרטרים ל-indexer, לדוגמא, נכתוב indexer שיחזיר לנו את המחרוזת הראשונה שמתחילה באות מסוימת ומסתימות באחרת, אם לא מנצא ערך נחזיר null:

```
public string this[char start, char end]
{
    get
    {
        foreach(string s in strings)
            if (s[0] == start && s[s.Length-1] == end)
                return s;
        return null;
    }
}
```

בהנחה שהמחרוזות לא ריקות.

18.3. פונקציות המרה

ב-C# ניתן לבצע המרה בין טיפוסים בסיסיים שונים על ידי casting.

נוכל גם לאפשר המרה בין אובייקטים שונים, ואף לפרט אם ההמרה תתבצע באופן אוטומטי (implicit) או באופן מפורש (explicit) באמצעות אופרטור המרה.

הצורה הדקדוקית להמרה היא כלהלן:

```
public static implicit operator TargetType(SourceType s)
{

}

public static explicit operator TargetType(SourceType s)
{

}
```

פונקציות המרה, בדומה לפונקציות החופפות אופרטורים, חייבות להיות פונקציות סטטיות.

נשים לב להבדל בין C# ל-C++: ב-C++ פונקציה בונה עם פרמטר אחד יכלה להיות פונקצית המרה.

ב-C# זה איננו המצב. אנו חייבים להגדיר באופן מפורש פונקצית המרה.

19. Delegates וטיפול באירועים

19.1. Delegate

Delegate הוא מעין מצביע לפונקציה. Delegate יכול לשמש כמצביע לפונקציה סטטית או פונקציה של אובייקט. באופן כזה מחלקות שונות יכולות להעביר ביניהן מצביעים אל הפונקציות שלהן.

הצורה הדקדוקית של delegate היא כלהלן:

```
delegate Return_Type DelegateName(parameters);
```

קוד זה יגדיר טיפוס חדש בשם DelegateName שיהווה מצביע לפונקציה שמקבלת פרמטרים כפי שמצוין בסוגריים, ומחזירה ערך מסוג Return_Type. ניתן להגדיר את הdelegate בתוך מחלקה או מחוץ לה.

דוגמא:

```
using System;

delegate void PrintFunc(string s);

class MainClass
{

	static void Print1(string s)
	{
		Console.WriteLine("****{0}****", s);
	}

	static void Print2(string s)
	{
		Console.WriteLine("----{0}----", s);
	}

	public static int Main()
	{
```

```
            PrintFunc prnFunc = new PrintFunc(Print1);
            prnFunc("Hello, World");
            prnFunc = new PrintFunc(Print2);
            prnFunc("Hello, World");
            return 0;
      }
}
```

הפלט שיתקבל יהיה:

```
****Hello, World****
----Hello, World----
```

דוגמא נוספת:

```
using System;

// delegate declaration
delegate void MyDelegate();

public class MyClass
{
      public void InstanceMethod()
      {
            Console.WriteLine("A message from the instance method.");
      }

      static public void StaticMethod()
      {
            Console.WriteLine("A message from the static method.");
      }
}

public class MainClass
{
      static public void Main()
      {
            MyClass p = new MyClass();

            // Map the delegate to the instance method:
```

```
            MyDelegate d = new MyDelegate(p.InstanceMethod);
            d();

            // Map to the static method:
            d = new MyDelegate(MyClass.StaticMethod);
            d();
        }
}
```

כאשר אנו יוצרים delegate אנו יוצרים למעשה אובייקט של המחלקה delegate.

למחלקה זו המאפיינים והפונקציות הבאים:

מאפיין / פונקציה	משמעות
Target	אם הפונקציה המוצבעת שייכת למחלקה, יוחזר שם המחלקה. אם הפונקציה הינה פונקציה סטטית, יוחזר NULL.
Method	מחזיר את שם הפונקציה שה-delegate מצביע עליה.
Combine()	מחבר שתי פונקציות כך שה-delegate יפעיל את שתיהן.
GetInvocationList()	הפונקציה מחזירה ווקטור של Delegates כשכל כניסה מייצגת מצביע לפונקציה שתופעל בהפעלת ה-Delegate.
Remove()	מופיעה מצביע לפונקציה מתוך הווקטור המוחזק.

דוגמא:

```
using System;
//Step 1. Declare a delegate with the signature of
//         the encapsulated method
public delegate void MyDelegate(string input);

//Step 2. Define methods that match with the signature
//         of delegate declaration
class MyClass1
{
      public void delegateMethod1(string input)
      {
            Console.WriteLine(
                  "This is delegateMethod1 and the input to the
method is {0}",
                  input);
      }
      public void delegateMethod2(string input)
      {
            Console.WriteLine(
                  "This is delegateMethod2 and the input to the
method is {0}",
                  input);
      }
}

//Step 3. Create delegate object and plug in the methods
class MyClass2
{
      public MyDelegate createDelegate()
      {
            MyClass1 c2=new MyClass1();
            MyDelegate d1 = new MyDelegate(c2.delegateMethod1);
            MyDelegate d2 = new MyDelegate(c2.delegateMethod2);
            MyDelegate d3 = d1 + d2;
            return d3;
      }
}
```

```
//Step 4. Call the encapsulated methods through the delegate
class MyClass3
{
	public void callDelegate(MyDelegate d,string input)
	{
		d(input);
	}
}
class Driver
{
	static void Main(string[] args)
	{
		MyClass2 c2 = new MyClass2();
		MyDelegate d = c2.createDelegate();
		MyClass3 c3 = new MyClass3();
		c3.callDelegate(d,"Calling the delegate");
	}
}
```

19.2. טיפול באירועים

טיפול באירועים בחלונות היא אחד הנושאים החשובים ביותר.

ב-C# היחידה המטפלת באירועים היא delegate.

נפתח בדוגמא:

```
using System;
namespace Events
{
	class Sink1
	{
		public int OnAdd(int x,int y)
		{

			Console.WriteLine("x+y={0}",x+y);
			Console.WriteLine("Input a number");
			return x+y;
		}

	}
	class Sink2
	{
		public int OnMul(int x,int y)
		{
			Console.WriteLine("x*y={0}",x*y);
			return x*y;
		}
		public int OnAvarage(int x,int y)
		{
			Console.WriteLine("Avarage x,y={0}",(x+y)/2);
			return (x+y)/2;
		}
	}
```

```
        class Test
        {
                public delegate int Func(int x,int y);
                public static event Func event1;
                public static event Func event2;
                static void ShowAllFunctions(Func e)
                {
                        Delegate[] arr=e.GetInvocationList();
                        Console.WriteLine("methods in event");
                        foreach(Delegate d in arr)
                        {
                                Console.WriteLine(d.Method.ToString());
                        }

                }
                static void Main(string[] args)
                {
                        Sink1 s1= new Sink1();
                        Sink2 s2= new Sink2();
                        event1 += new Func(s1.OnAdd);
                        event1+= new Func(s2.OnMul);
                        ShowAllFunctions(event1);
                        int z=event1(5,6);
                        Console.WriteLine("x={0}",z);
                        event2+=new Func(s2.OnAvarage);
                        event2(5,7);
                        event1-= new Func(s1.OnAdd);
                        event1(10,20);

                }
        }
}
```

בתוכנית הגדרנו שתי מחלקות המגדירות פונקציות שיגיבו ל-Events.

המחלקות הן Sink1 ו-Sink2.

הפונקציות הינן OnAdd, OnMul, OnAvarage.

כמו כן יצרנו שני events: event1, event2.

התוכנית מדפיסה למסך את הפונקציות הרשומות ל-Event בעזרת ()ShowAllFunctions ולאחר מכן מפעילה את ה-event.

בדוגמא הגדרנו את ה-events כסטטיים, אולם אין בכך הכרח.

20. טיפים לניהול זיכרון חכם

פרק זה נכתב על ידי **שושן כהן** ונערך על ידי **ניר אדר**.

20.1. האם ניהול הזיכרון לא מתבצע בשבילי?

כן, מנהל הזיכרון פועל מתחת לפני הצורך ולא חייבים להכיר אותו, אבל יש לשים לב למה שיפה ומושך אותי בסביבת net., ההתרחבות כלפי מעלה, ולא תנועה כלפי מעלה, כלומר האפשרות לחזור למטה, "לפרטים הקטנים", במקרי הצורך ועדיין להישאר בשליטה מלמעלה על "הפרטים הגדולים".

זאת שאלה שנשאלת בהרבה מאוד תחומים, האם זה הכרחי לדעת מה קורה מתחת למכסה המנוע כדי לנהוג במכונית? מתכנת לא באמת צריך לדאוג איך פועלת סביבת הריצה – או שאולי בעצם..?

אכן אפשר לנהוג במכונית מבלי לדעת איך פועל המנוע, אך אם ברצונך להיות נהג מרוצים מקצועי עליך לדעת כמה שרק אפשר על המכונית, או אם נחזור לעולמנו אנו, אם ברצונך לעסוק בפיתוח אפליקציות קטנות ופשוטות אין צורך בהתעמקות בתהליכי ניהול הזיכרון, אך אם ברצונך לעסוק בפיתוח אפליקציות גדולות ובעלות ביצועים קריטיים עליך "ללכלך את הידיים".

20.2. הדורות השונים

כדי להיות אפקטיבית בניהול הזיכרון סביבת net. מנסה להבין "מה קורה", ומה עליה לעשות תוך שמירה על ביצועים גבוהים וסריקת הזכרון בתדירות נמוכה. אובייקטים מסוימים עלולים להישאר כל משך זמן הריצה של התכנית בזכרון, ולעומתם יש כאלו שמשך חייהם מאוד קצר, לכן אם אובייקט חי זמן ארוך יותר, מתבצעת בדיקה אם יש אפשרות להיפטר ממנו לעיתים רחוקות יותר, ובכך נחסכים המשאבים הדרושים לבדיקה. התנהגות זו מושגת על ידי חלוקת הערימה (בה שמורים האובייקטים) לשלושה דורות, דורות 0, 1 ו-2. ניתן להמחיש את ההתנהגות על ידי המטאפורה הבאה:

- אתה מתבקש לטלפן לאיש קשר, אתה מוכשר בשינון מספרים ולכן אתה שומר את המספר "בראש" (הדור הראשון).
- אין תגובה ואתה צריך לשמור את המספר לזמן קצת ארוך יותר, בינתיים אתה מתבקש לטלפן למספר אנשי קשר נוספים. (הצורך לטלפן לאחרים הוא תנאי חשוב.)
- אתה מודע למגבלותיך, אתה מסוגל לשנן רק 10 מספרים שונים בו זמנית, אתה מגיע לנקודה קריטית בה אתה מחליט לרשום את המספרים שאתה עדיין זקוק להם על פתק (הדור השני).
- ככל שהיום חולף מספר הפתקיות הולך וגדל, והן מתחילות להסתיר את השולחן שלך.
- השולחן שלך התמלא עד אפס מקום, אתה מחליט לעשות סדר במהומה ומתחיל לעבור על הפתקיות, מחליט לאילו מהן אתה עדיין זקוק, אתה זורק את אלה שפג שימושן ואת השאר אתה רושם בספר הכתובות (הדור השלישי) לשהות יותר ממושכת.

כפי שציינתי בהתחלה, אובייקטים שרק נוצרו נכנסים לדור 0, הגודל של הדור הראשון נקבע לפי גודל המטמון של המעבד, הוא משתנה כתלות של קצב גדילת השימוש של התכנית בזיכרון. ברגע בו יתמלא הזיכרון של דור 0, יתבצע מעבר על האובייקטים השמורים בו אובייקטים שעבר זמנם יסומנו לפינוי עבור ה-GC, זה נקרא שלב הסימון והטאטוא (Mark and Sweep phase). כל האובייקטים שישרדו את השלב הזה יעברו דחיסה ויועברו לדור מספר 1, גם גודל דור 1 משתנה כתלות של קצב גדילת שימוש התכנית בזיכרון, וברגע שגם הוא יגמר יתבצע סימון וטאטוא גם בו, והאובייקטים השורדים יועברו לדור מספר 2, ואז יתבצע גם סימון וטאטוא גם עבור דור 0 (שכן דור 1 כבר פנוייה).

יחס בריא בין הדורות יוגדר כך: מספר האובייקטים בדור 0 יהיה 100, בדור 1 יהיה 10, ובדור 2 יהיה 1. פינוי רמה 0 בריאה ייקח בדרך כלל מספר אלפיות השנייה, פינוי רמה 1 יתבצע לאורך למעלה מ-30 אלפיות השנייה, ופינוי רמה שתיים לוקח זמן מה כתלות בתכנית.

20.3. ערימת האובייקטים הגדולים

נוסף על המבנה שהוצג ישנה גם ערימת האובייקטים הגדולים (LOH), בה שמורות אובייקטים בני למעלה מ-85,000 בתים, בערימה זו מתבצע איסוף בכל פעם שיש צורך במקטע חדש (ראה הסבר בהמשך), אך האובייקטים שבה לא נדחסים בשל גודלם והפגיעה שתתרחב בביצועים.

נשאלת השאלה: אם כל אובייקט בן 85,000 בתים מאוחסן ב-LOH, אין זה אומר שרוב האובייקטים יאוחסנו בה? – לא בהכרח, לדוגמא אובייקט טבלה יכול להכיל הרבה מאוד מידע, אך בעצם הוא מכיר רק מצביעים לאובייקטים אחרים, המידע השמור בתאים מאוחסן באובייקטים מסוג מחרוזת, כל עוד אלה לא גדולות מ-85,000 KB, לכן האובייקט לא יאוחסן בערימת האובייקטים הגדולים.

20.4. מקטעי זיכרון

בערימה מוקצה הזיכרון במקטעים, אשר גודלם תלוי בהגדרות: אם הוגדר <gcServer enabled="true"/>, יהיה גודלם 64MB, אחרת יהיה גודלם 32MB. בערימת האובייקטים הגדולים מוקצה הזיכרון במקטעים של 16MB. רק אובייקטים בדור 2 ובערימת האובייקטים הגדולים יכולים להשתרע על פני מספר מקטעים.

20.5. מה קורה בזמן איסוף זבל?

תהליך האיסוף, כולל מן ה-LOH:

- האובייקטים ב-LOH מסומנים: כל אוביקט נבדק, אם אין מצביעים אליו הוא מסומן כמוכן לאיסוף.
- ה-LOH מטואטא: כל האובייקטים שסומנו משוחררים מהזיכרון.
- ה-LOH **לא** עובר דחיסה.
- דור 2 עובר סימון.
- דור 2 עובר טאטוא.

- דור 2 עובד דחיסה (דמיינו הסרת מספר ספרים ממדף הדפרים שלכם, ודחיפתם כך שלא יושאר רווח, ובכך פינוי מקום בסוף המדף).
- דור 1 מסומן.
- דור 1 עובד טאטוא.
- דור 0 מסומן.
- דור 0 עובר טאטוא.

20.6. מספר טיפים

בשביל מה למדנו להכיר את התהליך והדורות אם לא בשביל לכלך את הידיים , אבל מה צריך לעשות ?
הנושא הזה יכול להיות מורחב הרבה יותר, אבל זה קצה הקרחון שנכנס למסמך זה, מבוטח אני שכל קוראי המסמך מסוגלים באם ירצו בכך, למצוא מקורות נוספים לעיון בנושא, אך הנה מספר הצעות מרכזיות.

20.6.1. הימנעו מדור 1

כן, כמובן שדור 1 טוב מדור 2, אך עליכם לכוון לשמירת מספר אובייקטים שנבחרו מראש לשמירה בדור 2, אמורים אלו להיות אובייקטים שמוגדרים בתחילת חייה של התכנית ומשוחררים אך ורק בסוף ריצתה. בעולם אידיאלי כל שאר המשתנים אמורים להיעלם כשם שבאו, ולעולם לא לעזוב את דור 0.

20.6.2. הימנעו מקריאה ל-GC.Collect()

כמעט לעולם אין לקרוא ל-GC.Collect באופן ידני, ובכמעט אני מתכוון פעם בחיים, לא פעם באפליקציה או כמובן לא פעם בקריאה לפונקציה. לעיתים יש צורך בקריאה ל-GC.Collect() לצורכי בדיקה האם הזיכרון שוחרר, באופן כללי לעולם לא תקראו לה. מנגנון איסוף הזבל מאזן את עצמו, קריאה לפונקציה זו הוא מעיין הפרה של איזון זה.

20.6.3. הימנעו מאובייקטים גדולים

אם ניתן לעשות זאת, השתדלו להישאר עם אובייקטים העומדים במגלת 85,000 הבתים, אם לא ניתן לעמוד בכלל זה, נסו להשתמש באובייקט אחד לאורך זמן ארוך שכן שימוש באובייקט אחד לאורך זמן ארוך עדיף משימוש באובייקטים רבים לאורך זמן קצר.

לדוגמא באפליקציית עיבוד תמונה בה נרצה לבצע מספר פעמים תהליך על תמונה, לדוגמא הדגשת קצוות, נשמור שתי תמונות (גלובאליות כמובן, הרי הן בדור 2) בנות אותו גודל ונקצה לשנייה זיכרון כאשר נטען את הראשונה, כך נוכל להסתמך כל התמונה המקורית נשארה קבועה מבלי לטעון אותה כל פעם מחדש (שחרור וטעינה כזה אובייקט גדול כל פעם משפיעה מאוד לרעה על הביצועים ועל כמות הזיכרון בשימוש), בכל עיבוד נקרא מן התמונה הראשונה, כך נוכל להסתמך כל התמונה המקורית נשארה קבועה מבלי לטעון אותה כל פעם מחדש (שחרור וטעינה כזה אובייקט גדול כל פעם משפיעה מאוד לרעה על הביצועים ועל כמות הזיכרון בשימוש), בכל עיבוד נקרא מן התמונה הראשונה נכתוב לשנייה, ולבסוף נציג את השנייה למשתמש.

20.6.4. הימנעו מפונקציות הורסות

כאשר לאובייקט שלך יש פונקציה הורסת היא תיקרא כאשר הוא כבר אינו בין החיים. עד כה הכל בסדר. לצערך האובייקט שלך יועבר לדור הבא משום שהוא אינו מוכן לאיסוף, משמע על באובייקטים עם פונקציות הורסות ימשיכו לדור 1 ולדור 2, ללא סיבה מוצדקת.

21. סיום

בזאת מגיע לסיומו המסמך על שפת #C. למרות שניסינו לכסות כאן מגוון רחב של נושאים, זוהי רק טעימה מהשפה והדרך להתמחות בה עוברת דרך כתיבה של הרבה קוד והתמקצעות.

מספר נושאים חשובים שלא הוזכרו במסמך זה ששווים אזכור:

- **Windows Forms** – כיצד מעצבים את התוכנות שלכם כך שיראו כמו כל תוכנת Windows פופולרית? התשובה – בעזרת אוסף המחלקות והפקדים של Windows Forms. הרשת מלאה בחומרים שיוכלו לתת לכם כיוון, ובנוסף עורכים רבים הופכים את תהליך יצירת הממשק לאינטואיטיבי ביותר.
- **Serialization, XML** – שמירת המידע אותו מייצרת התוכנית באמצעות XML הינו אחד האמצעים החשובים שניתן להשתמש בהם כדי לשמור את המידע והאובייקטים של התוכניות אותן נכתוב בין ריצה לריצה. התחביר אינו מסובך, והיתרונות גדולים.
- **אינטרנט ותקשורת** – אפליקציות רבות משתמשות ברשת האינטרנט. שפת #C כוללת כחלק מהשפה כלים רבים ליצירת אפליקציות מבוססות אינטרנט.

ה-CLR כולל עוד כמות עצומה של מחלקות, מחלקות המבצעות מניפולציות מתוחכמות על טקסט, מחלקות הקשורות לתחום האבטחה ועוד מגוון רחב של מחלקות. פורומים מקצועיים באינטרנט, אתרים העוסקים בשפה וכדו' הינם מקורות עשירים שיכולים לעזור לך להתמקצע בכיוונים הדרושים לך.

בהצלחה בהמשך דרכך עם השפה!

ניר אדר שושן כהן