בס"ד

סיכום לשפת C#

const- ערך קבוע שלא משתנה, נקבע בזמן הקומפילציה. מכיוון שהקומפיילר לא יודע איזה ערך יהיה בפונקציה, הוא לא יכול להגדיר ערך const למשתני פונקציה. Const הוא סטטיק אוטומטי.

readonly- בדומה לconst, זה ערך קבוע שלא משתנה, אך הוא יותר דינאמי, בדומה לref למקום בזכרון. לכן ניתן להשתמש בו בפונקציות.

טיפוסים פרמיטיביים- הטיפוסים הפרימיטביים בc# כדוגמאת int, double, bool, long, short. הם אובייקטים היורשים מvaluetype, לכן הם יוצרים מופע (אינסטנס) של האובייקט ולא רפרנס. לכן הם יושבים במחסנית הקריאות ולא בערימה הדינמית.

Var- משתנה כללי שמשאיר לקומפיילר להסיק מתוכן המשתנה, איזה סוג הוא.

ירושה וממשקים- ממשקים יסומנו באות I. כך ניתן להבחין בין ממשק למחלקה אבסטרקטית. מסמנין נקודותיים אחרי שם המחלקה וכותבים איזה מחלקה נירש ממנו או את איזה ממשק נממש, עם פסיק בין כל שם מחלקה/ממשק שנממש.

ההבדל בין interface למחלקה אבסטרקטית- אינטרפייס נועד כדי ליצור ממשק זהה לכמה אובייקטים שונים (אפשר להגדיר את זה כחוזה ביניהם). אינטרפייס או ממשק, אינו יכול להכיל משתנים, אלא רק הצהרות לא ממומשות לפונקציה.
לעומת זאת מחלקה אבסטרקטית יכול להכיל כל דבר שמחלקה רגילה יכולה להכיל, וניתן להשתמש בה בעת הצורך כשאנו רוצים לחסוך כתיבת קוד מיותר. לכן היא יכולה להחשב כמחלקת בסיס יותר מאשר כממשק, אך היא לא משתמשת בעקרון הפולימורפיזם בשונה מממשק.

struct- מבנה ושדותיו הם internal (זמינים רק למחלקות שבאותו פרוייקט) כברירת מחדל.
מבנה לא מחייב יצירת מופע שלו כ-new, לכן ניתן להשתמש במבנה מבלי להשתמש בערימה ובכך לחסוך זכרון.
במקרה כזה נצטרך לאתחל כל אחד משדות המבנה בצורה אוטומטית.
אין הורשה במבנים.
מבנה הוא טיפוס value type.

Set ו-get-

```
Class foo
{
        public int value { get; internal set;}
}
In code:
Foo foo = new Foo () { value=42 };
```

אבל רגע! היופי בsetter ו-getter -של java שניתן להוסיף להן קוד שירוץ עם הקריאה למתודה. ניתן לעשות זאת גם בסי שארפ! כברירת מחדל הם יתנהגו כמו שדות אך ניתן לתת להם יישום מיוחד

```
public class
{
        private int value;
```

```
        public int Value {
                get { return this.value; }
                set {
                // the word "value" here that you're setting to the field
                 // "this.value" is actually a C# keyword available in
                 // all setters.
                this.value = value; }
        }
        public bool IsEven {
                get { return this.value % 2 == 0; }
        }
}
```

<u>אופרטורים-</u> ניתן לדרוס אופרטורים בשביל מחלקות כמו בcpp.

```
public struct Test {
        public int i ;
        public int j;
         public Test(int _i, int _j) { i = _i; j = _j; }
        public static Test operator +(Test t1,Test t2) {
                Test t3;
                t3.i = t1.i + t2.i;
                t3.j = t1.j + t2.j;
                return t3;
                }
 }
```

העמסת אופרטור אונרי:

```
public static Circle operator ++(Circle c)
{
```

```
//      מימוש
}
```

מבנה כותרת ההעמסה של **אופרטור אונרי** יהיה בדיוק בסדר הבא:

1. **public static**
2. הערך **המוחזר** (Circleבדוגמא הנ"ל)
3. מילת המפתח **operator**
4. **האופרטור** שרוצים להעמיס (++ בדוגמא)
5. **פרמטר** מטיפוס המחלקה(Circle c בדוגמא)

העמסת אופרטור בינארי:

```
public static Circle operator +(Circle c, int x)
{
//      מימוש
}
```

מבנה כותרת ההעמסה של **אופרטור בינארי** יהיה בדיוק בסדר הבא:

1. **public static**
2. הערך **המוחזר** (Circleבדוגמא הנ"ל)
3. מילת המפתח **operator**
4. **האופרטור** שרוצים להעמיס (++ בדוגמא)
5. **פרמטר** מטיפוס המחלקה(Circle c בדוגמא)
6. **פרמטר** נוסף, מטיפוס כלשהו (int xבדוגמא)

מתודות וירטואליות- כמו בCPP יש לרשום virtual בחותמת הפונקציה במידה ונרצה שמחלקת הבן תוכל לדרוס את פונקציית מחלקת האב. וכדי שאכן הפונקציה תדרס נרשום override.

static באובייקטים אל מול static במחלקות- המילה השמורה static באובייקט מחלקה אומר שהאובייקט הזה משותף לכל האובייקטים מאותו סוג. (נקדם אותו דרך אובייקט אחד, יקודם בכל שאר האובייקטים).
במחלקות בג'אווה כשרצינו ליצור מחלקה בתוך מחלקה השתמשנו בstatic לחתימה של המחלקה הפנימית כדי שיהיה ניתן לגשת אליה דרך המחלקה העוטפת אותה.
בC# אין צורך, זה מוגדר אוטומטית.

מחלקה סטטית

ניתן להגדיר גם static class, המשמעות היא:כל מה שנמצא במחלקה **חייב** להיות static (משתנים, פונקציות, מאפיינים)...**לא** ניתן ליצור מופעים (אובייקט) מהמחלקה, **לא** ניתן לרשת מהמחלקה.

params- מילה שמורה המאפשרת לפונקציה לקבל מספר מסויים של פרמטרים ובתוך הפונקציה להשתמש בה כlist של ערכים. דוגמא:

```
public static void UseParams2(params object[] list)
    {
```

```
        for (int i = 0; i < list.Length; i++)
        {
            Console.Write(list[i] + " ");
        }
        Console.WriteLine();
    }

static void Main()
    {
        // You can send a comma-separated list of arguments of the
        // specified type
        UseParams2(1, 'a', "test");

}
```

ניתן להגביל את הרשימה לאובייקט מסוג מסויים על ידי שינוי המשתנה object למתנה אחר. לדוגמא int.

-ref כששולחים לפונקציה פרמטרים, הפונקציה יוצרת העתק של הפרמטר ולא משתמשת בתאים המקוריים של האובייקט.
כשאנו רוצים להשתמש במקום המקורי בזכרון, נשתמש במילה שמורה ref. גם בחתימת הפונקציה, וגם לפני שם האובייקט שנשלח כפרמטר לפונקצייה.

-out נשאלת השאלה, מה נעשה אם נרצה שפונקציה תחזיר יותר מערך אחד?

לצורך כך נוכל להגדיר פרמטר של הפונקציה בתור פרמטר **out**, ע"י הוספת המילה **out** לפני הגדרת הטיפוס שלו.
פרמטר שמוגדר בתור **out** הוא פרמטר שמועבר ע"י הפונקציה הקוראת (Main) רק לצורך קבלת ערך החזרה.

לדוגמא, קטע הקוד הבא מגדיר פונקציה שיש לה גם ערך החזרה וגם שני פרמטרים מטיפוס **out**:

```
static string GetMultipleValues(out string firstName, out string
lastName)
{
    Console.Write("Enter first name: ");
    firstName = Console.ReadLine();

    Console.Write("enter last name: ");
    lastName = Console.ReadLine();

    return firstName + " " + lastName;
}
```

פונקציה זו קולטת מהמשתמש ערכים לתוך הפרמטרים firstName ו lastName ובנוסף מחזירה ערך שמכיל את השם המלא.

בעת הקריאה לפונקציה עם פרמטר מטיפוס **out** יש לציין את המילה **out** לפני שם המשתנה:

```
static void Main(string[] args)
{
    string full, first, last;
    full = GetMultipleValues(out first, out last);
}
```

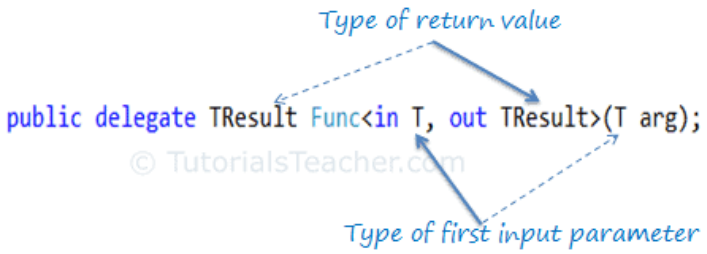
אנו קוראים לפונקציה GetMultipleValues ומעבירים לה את הפרמטרים first ו last תוך הוספת המילה **out** לפני.

המשתנה full יקבל את ערך ההחזרה הרגיל של הפונקציה ואילו המשתנים first ו last יקבלו את הערכים שיקלטו מהמשתמש.

func delegate מייצגים של פונקציה- טיפוס מצביע לפונקציה. המשתנה הזה יוגדר עם inputs ו output מוגדר מראש, וכל פונקציה העומדת בתנאים של הdelegate יוכלו להיות מיוצגים על ידו.

ניתן לרשום delegate באופן הבא:

```
namespace System
{
    public delegate TResult Func<in T, out TResult>(T arg);
}
```




```
class Program
{
    static int Sum(int x, int y)
    {
        return x + y;
    }

    static void Main(string[] args)
    {
        Func<int,int, int> add = Sum;

        int result = add(10, 10);

        Console.WriteLine(result);
    }
}
```

בס"ד

אפשרי גם לרשום פונקציה אנונימית ולחסוך יצירת פונקציה:

```
Func<int> getRandomNumber = delegate()
                            {
                                Random rnd = new Random();
                                return rnd.Next(1, 100);
                            };
```

או על ידי lambda:

```
Func<int, int, int>  Sum  = (x, y) => x + y;
```

אך זה אינו השימוש הנפוץ ביותר לdelegate. על ידי delegate אפשר לייצג מספר רב של פונקציות דרך משתנה אחד. זה נקרא **Action delegate**, ומשתמשים בו לתכנון נכון של מחלקה עם מספר רב של פונקציות שונות הקשורות בדרך כלשהי אחת בשניה:

דוגמא:

```
namespace Delegates
{
  delegate int Calc(int x, int y);

  class Program
  {
    static int Add(int a, int b)
    {
      Console.WriteLine("Add");
      return a + b;
    }

    static int Sub(int a, int b)
    {
      Console.WriteLine("Sub");
      return a - b;
    }

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

```
      Calc c;
      c = new Calc(Add);
      c += new Calc(Sub);
      int result = c(4, 6);

      Console.WriteLine(result);
    }
  }
}
```

דוגמא ללמה צריך delegate:

```
namespace SortWithDelegate
{
  public delegate int CompareDeleg(int a, int b);

  class Program
  {
    static int[] arr;

    static int CompareAsc(int x, int y)
    {
      return x - y;
    }

    static int CompareDesc(int x, int y)
    {
      return y - x;
    }

    static void Sort(CompareDeleg compareMethod)
```

```
{
  for (int i = 0; i < arr.Length - 1; i++)
    for (int j = i + 1; j < arr.Length; j++)
      if (compareMethod(arr[i], arr[j]) > 0)
        Replace(i, j);
}

private static void Replace(int i, int j)
{
  int tmp = arr[i];
  arr[i] = arr[j];
  arr[j] = tmp;
}

static void PrintArr()
{
  foreach (int num in arr)
    Console.Write(num + ",");

  Console.WriteLine();
}

static void Main(string[] args)
{
  arr = new int[] { 123, 200, -63, 2, 7612, -13, 8 };
  Console.WriteLine("Orginal nuumbers:");
  PrintArr();

  Sort(new CompareDeleg(CompareAsc));
  Console.WriteLine("\nAscending order:");
  PrintArr();

  Sort(new CompareDeleg(CompareDesc));
  Console.WriteLine("\nDescending order:");
  PrintArr();
}
```

```
    }
}
```

בתוכנית זו ישנה פונקציה בשם Sort שתפקידה למיין מערך מסוג .int[] חישבו שהיינו רוצים למיין את המערך בצורות שונות פעם בסדר עולה ופעם בסדר יורד, עם הידע שיש לנו עד היום היינו צריכים לבנות 2 פונקציות, אחת למיון עולה ואחת למיון יורד. ואם בעתיד היינו רוצים שיטת מיון אחרת (חישבו למשל על תאריכים שאפשר למיין בשיטות שונות: שנים, רבעונים, חודשים ועוד).

הרעיון הוא שהפונקציה Sort תקבל את המערך ו delegate -בשם CompareDeleg שיטפל בצורת המיון, **כך שהפונקציה לא תגביל לסוג מיון מסוים אלא תיתן למי שמפעיל אותה לקבוע את סוג המיון.**

<u>משתנה Action-</u> משתנה שמכיל בתוכו פונקציה שניתנת להפעלה שהיא void ולא מקבלת משתנים.

<u>משתנה Predicate-</u> משתנה שמחזיק פונקציה שמחזירה בוליין.

<u>LINQ Queries:</u>

אופרטורי השאילתות בחבילה זו עובדים עם אובייקטים מסוג lenumarable וIQuearyable. לכן אפשר להגיד שיש שני סטים של שאילתות שכל אחד פועל על אחת מהמחלקות המוזכרות למעלה.

<u>Aggregate:</u> מבצעים על אובייקט שהוא lenumerable, זוהי פעולה שדרכה אפשר מרשימה אחת ליצור output שהוא רשימה אחרת המכילה תוצאה של פונקציה שכתבנו.

<u>Where clause:</u> אפשר להשתמש בשאילתות דומות לsql על טיפוסים שהם lenumarable או מסדי נתונים שונים בכדי להוציא מידע העומד בתנאים מסויימים.
ניתן להגדיר שאילתה בדרך הבאה:

```
Int [] array= For item In otherArray Where item>50 && item<100 select item;
```

בעצם בחרנו את כל האיברים מתוך המערך השני שהם גדולים מ50 אך קטנים מ100.

אפשר לרשום את זה כך: int []
```
array=otherArray.Where(model=>model>50).Where(model=>model<100);.
```
נשים לב כי model היא מילה שמורה. (לא בטוח נכון)

בס"ד

```
string sentence = "the quick brown fox jumps over the lazy dog";
// Split the string into individual words to create a collection.
string[] words = sentence.Split(' ');

// Using query expression syntax.
var query = from word in words
            group word.ToUpper() by word.Length into gr
            orderby gr.Key
            select new { Length = gr.Key, Words = gr };

// Using method-based query syntax.
var query2 = words.
    GroupBy(w => w.Length, w => w.ToUpper()).
    Select(g => new { Length = g.Key, Words = g }).
    OrderBy(o => o.Length);

foreach (var obj in query)
{
    Console.WriteLine("Words of length {0}:", obj.Length);
    foreach (string word in obj.Words)
        Console.WriteLine(word);
}

// This code example produces the following output:
//
// Words of length 3:
// THE
// FOX
// THE
// DOG
// Words of length 4:
// OVER
// LAZY
// Words of length 5:
// QUICK
// BROWN
// JUMPS
```

Anonymous Types- אובייקט (או מחלקה) ללא שם שהמשתנים שלו הם read-only וגם public. לא יכול להכיל שום דבר אחר, לא פונקציה אנונימית וכו'. וכן לא ניתן לאתחל את המשתנים ב null או pointer type.

```
var student = new { Id = 1, FirstName = "James", LastName = "Bond" };
Console.WriteLine(student.Id); //output: 1
Console.WriteLine(student.FirstName); //output: James
Console.WriteLine(student.LastName); //output: Bond

student.Id = 2;//Error: cannot chage value
student.FirstName = "Steve";//Error: cannot chage value
```

Example: Array of Anonymous Types

```
var students = new[] {
            new { Id = 1, FirstName = "James", LastName = "Bond" },
            new { Id = 2, FirstName = "Steve", LastName = "Jobs" },
            new { Id = 3, FirstName = "Bill", LastName = "Gates" }
    };
```

בס"ד

Example: LINQ Query returns an Anonymous Type

```
class Program
{
    static void Main(string[] args)
    {
        IList<Student> studentList = new List<Student>() {
            new Student() { StudentID = 1, StudentName = "John", age = 18 },
            new Student() { StudentID = 2, StudentName = "Steve",  age = 21 },
            new Student() { StudentID = 3, StudentName = "Bill",  age = 18 },
            new Student() { StudentID = 4, StudentName = "Ram" , age = 20  },
            new Student() { StudentID = 5, StudentName = "Ron" , age = 21 }
        };

        var students = from s in studentList
                       select new { Id = s.StudentID, Name = s.StudentName };

        foreach(var stud in students)
            Console.WriteLine(stud.Id + "-" + stud.Name);
    }
}
```

typeOf- שאילתא שדרכה אפשר להשיג את כל המשתנים ברשימה שהם מסוג מסויים. למשל:

```
C#

System.Collections.ArrayList fruits = new System.Collections.ArrayList(4);
fruits.Add("Mango");
fruits.Add("Orange");
fruits.Add("Apple");
fruits.Add(3.0);
fruits.Add("Banana");

// Apply OfType() to the ArrayList.
IEnumerable<string> query1 = fruits.OfType<string>();

Console.WriteLine("Elements of type 'string' are:");
foreach (string fruit in query1)
{
    Console.WriteLine(fruit);
}

// The following query shows that the standard query operators such as
// Where() can be applied to the ArrayList type after calling OfType().
IEnumerable<string> query2 =
    fruits.OfType<string>().Where(fruit => fruit.ToLower().Contains("n"));

Console.WriteLine("\nThe following strings contain 'n':");
foreach (string fruit in query2)
{
    Console.WriteLine(fruit);
}

// This code produces the following output:
//
// Elements of type 'string' are:
// Mango
// Orange
// Apple
// Banana
//
// The following strings contain 'n':
// Mango
// Orange
// Banana
```