

פולימורפיזם¹, ממשקים והשוואת עצמים

מאמר זה דן בשני תת נושאים בתחום פולימורפיזם וממשקים. תחילה נדון בהגדרות הנדרשות למימוש פולימורפיזם, ונראה מהן ההשלכות כאשר כותרת פעולה במחלקה היורשת נכונה תחבירית אבל לא לוגית. נראה גם כיצד 'לשבור' התנהגות פולימורפית במכונן. כלומר נדון ב-

I. פעולות המוגדרות בעזרת הרשאות: `new` או `override`, `virtual`

בחלק השני, נדון באוסף הפעולות וממשקים להשוואה בין עצמים. מכיוון שישנם מספר ממשקים החושפים פעולות מוגדרות כפעולות להשוואה בין עצמים, לכל אחת מהפעולות מוגדר תפקיד ייחודי. הפעולות והממשקים בהן נדון הם:

II. פעולה `Equals()` מממשק `IEquatable`,

III. פעולה `Comparable()` מממשק `IComparable`,

IV. ופעולה `Compare()` מממשק `IComparer`

למאמר זה מצורף פרויקט בויז'ואל סטודיו שבו מוגדרות מספר מחלקות. במחלקות אילו ניתן למצוא דוגמאות לנושאים מתוכן המאמר². הדוגמאות בשפת `C#`.

New - I³Virtual, Override .I

פולימורפיזם דינמי - המילה השמורה `virtual` נועדה לציין שניתן לממש פעולה בעלת כותרת זהה במחלקה יורשת כך שהפניה לעצם מטיפוס מחלקת הבסיס כתוצאה מהמרה כלפי מעלה, יפנה זימון של פעולות למימושן במחלקה היורשת. היתרון הוא שניתן לפנות למופעים של עצמים ממחלקות יורשות (מאותה מחלקה אב) שונות, דרך הפניה לעצם ממחלקת הבסיס, כך שזימון הפעולות יהיה אחיד והפעולה המתאימה למופע תזומן. היתרון התחבירי בולט כאשר יש צורך באוסף של עצמים (כמו בתוך רשימה, מערך וכו') שלכולם מחלקת בסיס זהה.

הדוגמא הקלאסית היא הפעולה

```
public virtual string ToString()
```

הפעולה הזאת מוגדרת במחלקה `Object` שכל עצם יורש ממנה. כל מחלקה יכולה לממש את הפעולה הזו – בעזרת המילה השמורה `override` – כך:

```
public override string ToString()
```

כמובן פעולות אחרות המוגדרות במחלקה יכולות גם כן להיות מוגדרות בעזרת ההרשאה `virtual`.

כדוגמה נוספת (ראו קוד בפרויקט המצורף), במחלקות `Person`, `Teacher` ו-`SchoolMaster` הפעולה `GetSalary()` הוגדרה כ-`virtual` במחלקת הבסיס `Person`.

¹ יש להבדיל בין שני הסוגים של פולימורפיזם: פולימורפיזם פרמטרי/סטטי (בזמן קומפילציה), ופולימורפיזם הכללה/דינמי (בזמן ריצה). מאמר זה דן בפולימורפיזם בזמן ריצה. פולימורפיזם סטטי נוצר כאשר אותה פעולה מוגדרת יותר מפעם כאשר הן שונות ברשימת הפרמטרים שהן מקבלות. כך שבזמן הקומפילציה ניתן לדעת איזו גרסה של הפעולה מזומנת. פולימורפיזם דינמי (function overloading) יזמן את הפעולה המתאימה למופע שעברו זמנה הפעולה. ² חלקן של המחלקות לא ממומשות. מחלקות אילו נועדו לשמש כבסיס להרחבת הנושא ו/או כגיוון למורה.

סוף מצא

```
namespace ConsoleApp1
{
    2 references
    class Person
    {
        string LastName, FirstName;
        double Salary;
        1 reference
        public Person(string LastName, string FirstName, double Salary)
        {
            this.LastName = LastName;
            this.FirstName = FirstName;
            this.Salary = Salary;
        }
        1 reference
        public virtual double GetSalary()
        {
            return this.Salary;
        }
    }
}
```

במחלקות היורשות ישנן שלוש אפשרויות להגדרת ומימוש הפעולות שהוגדרו כ-virtual במחלקת הבסיס. שלוש סוגי ההרשאות הפולימורפיות האפשריות הן:

1. כלום –

```
1 reference
class Teacher : Person
{
    int Vetek;
    0 references
    public Teacher(string LastName, string FirstName, int Salary, int Vetek) : base(LastName, FirstName, Salary)
    {
        this.Vetek = Vetek;
    }
    0 references
    public double GetSalary()
    {
        double multipl;
        if (this.Vetek > 0)
            multipl = 1.5;
        else if (this.Vetek < 0)
            multipl = 0.5;
        return multipl * base.GetSalary();
    }
}
```

ללא הרשאה ספציפית, ויז'ואל סטודיו מייצר הזהרת תחביר המבקשת הבהרה: האם הכוונת המשורר למימוש כהרשאת new או כהרשאת override (הסבר בהמשך). בריחת המחדל תהיה הרשאת - new.

2. **override** – כלומר הפעולה במחלקה היורשת תזומן גם כאשר הזימון הוא דרך הפניה מטיפוס מחלקת הבסיס (Person) כל עוד העצם הוא מופע של טיפוס ממחלקה היורשת מ-Person. הפעולה במחלקה היורשת 'דורסת' את הפעולה במחלקת הבסיס. משתמע מכך ש:

- לא ניתן לזמן את הפעולה הנדרסת במחלקת הבסיס בעזרת הפניה לעצם מהמחלקה היורשת מחוץ למחלקה;
- בכדי לגשת לפעולה במחלקת הבסיס מתוך המחלקה היורשת יש לציין גישה במפורש בעזרת התחביר: `base.GetSalary()`. (ראו דוגמות בקוד)

```

    }
    public override double GetSalary()
    {
        double multiplier = 1.0;
        if (Vetek > 20)
            multiplier = 1.5;
        else if (Vetek > 10)
            multiplier = 1.25;
        return multiplier * base.GetSalary();
    }
}

```

```

class Car: Vehicle
{
    public override void VirtualFunc_Override()
}

```

3. **New** – מבטל (חוסם) את ההתנהגות הפולימורפית של הפעולה. זימון הפעולה GetSalary() מהפניה של Person שנוצרה מהמרה כלפי מעלה ממופע של Teacher יזמן את הפעולה במחלקת הבסיס ולא את זו במחלקת Teacher

```

class Teacher : Person
{
    int Vetek;
    public Teacher(string LastName, string FirstName, int Salary, int Vetek) : base(LastName, FirstName, Salary)
    {
        this.Vetek = Vetek;
    }
    public new double GetSalary()
    {
        double multiplier = 1.0;
        if (this.Vetek > 20)
            multiplier = 1.5;
        else if (this.Vetek > 10)
            multiplier = 1.25;
        return multiplier * base.GetSalary();
    }
}

```

הערה – במחלקות יורשות שבהן ממומשת פעולה שהוגדרה כ- virtual במחלקת הבסיס, אין צורך להגדיר את הפעולות הדורסות כ- virtual. דוגמאות נוספות במחלקת Vehicle בפרויקט המצורף.

ממשקי השוואה

חשיבות ממשקי השוואה נובעת מכך שהם משמשים בסיס לפעולות מיון. הדגש לפעולות – ברבים – מכיוון שניתן לכתוב פעולות השוואה שונות ובכך ליצור סדרי ממוין שונה בהתאם לדרישות הלוגיות של התוכנית. ישנם שני 'סוגי' השוואה. השוואה `obj1.Equals(other)` שהתוצר השוואה הוא ערך בוליאני אמת (`true`) – כאשר השני המופעים הכן שווים (זהים); ושקר (`false`) אחרת. והשוואה רלטיבית – שימושית למיון - `obj1.CompareTo(other)` או `Compare(obj1, obj2)` שהתוצר שלהם:

- מספר שלם שלילי כאשר מקומו של `obj1` לפני `obj2` בסדר ממוין
- 0 כאשר מקומם של `obj1` ו- `obj2` זהה (חלופי) בסדר ממוין
- מספר שלם חיובי כאשר מקומו של `obj1` אחרי `obj2` בסדר ממוין

ממשק ופעולה `IEquatable.Equals()`

.ii

הפעולה `Equals()` מוגדרת כפעולה וירטואלית במחלקה `Object`. הפעולה מאפשרת למופע לבדוק אם הוא 'זהה' למופע אחר. פה צריך להבדיל בין המונחים 'זהה' ו'שווה ערך'.
 ○ כשמדובר במופעים של ערכיים, ניתן לומר שהם `Equals` כאר הם מאותו טיפוס (או שניתן להמיר אחד לשני ללא אובדן מידע), ויש להם את אותו הערך, לדוגמא:

```
int i = 7, j = 7;
double d = 7;
i.Equals(d); // false
d.Equals(i); // true. Upcast i to double is OK.
j.Equals(i); // true
i.Equals(j); // true
```

- כשמדובר בהפניות הפעולה `Equals` זהה לפעולה `ReferenceEquals`.
- כאשר המופעים הם מחרוזות, בגלל שהם `Immutable`, השוואה נעשית על התווים למרות שמדור בהפניה.

`public override bool Equals(object other)`

כותרת הפעולה:

מימוש מחלקה (הגדרת אובייקט חדש מסוג הפנייה) אינה מחויבת לממש את `Equals`. במידה והפעולה אינה ממומשת במחלקה, תזומן הפעולה של מחלקת הבסיס `Object.Equals()`. שני מופעים של עצמים יהיו `Equals` כאשר ההפניות שלהם זהות. ויהיו לא `Equals` - גם עם כל התכונות שלהם זהות - כאשר אין להם את אותן ההפניות. (ראו [MSDN Equals](#))

יש להיזהר מכיוון שנהוג (ללמד) לדרוס את `Equals` למען מימוש שווה תוכן ולא שווה הפניה. C# מאפשרת את זה ומימוש יכול להחזיר שוויון בהתאם לקריטריונים של המתכנת שיכללו השוואה של חלק או כל התכונות של המופעים. אך זה שימוש מסוכן בממשקי השוואה. מהגדרת המובנת של הפעולה, כך שהיא מחזירה `true` לשני מופעים, נובע ששינוי בערך של תכונה של אחד מהם ישנה גם את ערך התכונה של השני! אבל כאשר מממשים את `Equals` להשוות תוכן, ולא להשוואת הפניות, שינוי תכונה של אחד מהם לא ישנה את ערך התכונה של השני. התוצאות יכולות להיות קטסטרופליות. בדוגמא הבאה העלאת משכורת יכולה להתבצע על ה- `Person` שלא נכון מכיוון ש `Equals` השווה תכונות ולא הפניות.

```
public class Person
{
```

```
private string LastName;
private int Salary;
public Person(string LastName, int salary)
{
    this.LastName = LastName;
    this.salaty = Salary;
}
public void IncreaseSalary(int RaiseAmount)
{
    this.Salary += RaiseAmount;
}
public override bool Equals(Object Other)
{
    Person p = Other as Person;
    If (p == null)
        return false;
    return this.LastName.Equals(p.LastName);
}
}
```

III. ממשק ופעולה IComparable.CompareTo()

כאמור ממשק זה מיועד כהשוואה ממינית , והפעולה מחזירה אחד משולה ערכים: שלילי, 0, או חיובית בהתאם לסדר המיון בין העצם עליו מתבצעת ההשוואה לבין העצם האחר. מהדוגמה הבאה, ניתן לראות שאפשר לממש את CompareTo כך שתתבצע השווה מורכבת ככל שנרצה שאינה חייבת להיות מבוססת רק על ערך של תכונה אחת של העצמים.

כותרת הפעולה: `public int CompareTo(object other)`

```
public class Car: IComparable
{
    public string Name;
    public int MaxSpeed;
    public Car(string Name, int MaxSpeed) { this.Name = Name; this.MaxSpeed = MaxSpeed; }
    public int CompareTo(object obj)
    {
        if (!(obj is Car))
        {
            throw new ArgumentException("Compared Object is not of car");
        }
        Car other = obj as Car;
        // use string.CompareTo for sorting by name
        return this.Name.CompareTo(other.GetName());
    }
}
```

```
// alternatively, we could sort by MaxSpeed

public int CompareTo(object obj)
{
    if (!(obj is Car))
    {
        throw new ArgumentException("Compared Object is not of car");
    }
    Car car = obj as Car;
    // use int.CompareTo for sorting
    return MaxSpeed.CompareTo(car.GetMaxSpeed());
}
```

הערה – מיון בסדר הפוך ניתן לקבל ע"י שינוי שורה אחת בקוד:

```
return other.Name.CompareTo(this.GetName());
```

ממשק זה משמש לאובייקטים מסוג Collection כך שיוכלו למיין את האיברים באופן. לדוגמה:

```
Car[] cars = {new Car("Ford", 80), new Car("Golf", 90), new Car("Corolla", 85) };
Array.Sort(cars);

Person של לדוגמה

public int CompareTo(object obj)
{
    Person other = obj as Person;
    if (other == null)
        return -1;
    return this.name.CompareTo(other.GetName());
}

//
// Main
//
Person[] people = { new Person("John"), new Person("Bob"), new Person("Joshua"),
                    new Person("Alice") };

Array.Sort(people);
```

IV. ממשק IComparer.CompareTo()

ישנם מצבים בהם המחלקה אינה מממשת את ממשק IComparable. איך נוכל למיין מופעים באוספים? על מנת לפתור את בעיה זו נשתמש בממשק IComparer. ממשק זה לא ממומש ע"י המחלקה עצמה. אבל (כאשר ממומש) ככלי עזר למיון הוא נגיש לפעולות המיון שמקבלות עצם מסוג IComparer כארגומנט. במחלקה נפרדת זו נממש את הפעולה Compare. אחד היתרונות שנובע מהחצנת הפעולה ההשוואה היא שאפשר ליצר מספר בלתי מובל של מחלקות השוואה שכול אחת מממשת קריטריונים השוואה שונים. למשל נוכל ליצר השוואה לפי מחיר רכב, והשוואה אחרת לפי שנת יצור, וכמובן השוואות על יותר מתכונה אחת של העצמים. גם פעולת Compare מחזירה שלושה ערכים כמו שהחזירה CompareTo.

```
public int Comparer(object first, object second)
```

נותרת הפעולה:

[דוגמה שלמה נמצאת כאן](#)

```
public class CompareByPrice : IComparer
{
    public CompareByPrice() { }
    //Implementing the Compare method
    public int Compare(object object1, object object2)
    {
        Car c1 = (Car) object1;
        Car c2 = (Car) object2;
        return int.Compare(c1.GetPrice(), c2.Getprice());
    }
}
public class CompareByManufactureDate : IComparer
{
    public CompareByManufactureDate() { }
    //Implementing the Compare method
    public int Compare(object object1, object object2)
    {
        Car c1 = (Car) object1;
        Car c2 = (Car) object2;
        return int.Compare(c1.GetDate(), c2.Date());
    }
}

//
// main
//
CompareByPrice cbp = new CompareByPrice();
Array.Sort(cars, cbp);
```

לסיכום עומדים לשרותינו שלושה כלים להשוואת מופעים. לכל אחד תפקיד שונה וגמישות שימוש שונה. Equals מבצע השוואה בסיסית בין מופעים ואינו מיעד לשימוש במיונים. CompareTo מיועד לשימוש במיונים אך המשתמש שאין לא גישה למחלקה עצמה אינו יכול לשלוט בקריטריונים המיון. והפעולה Compare שמחצינה את קריטריונים המיון מן העצמים שעבורם נעשית ההשוואה נותנת גמישות רבה למתכנת.