

לעיתים נרצה לבנות מחלקות אשר מאד דומות בתבניתם, ושונות רק בסוג הנתונים שהם מחזיקות.

דוגמאות:

- מחלקה לניהול רשימה של נתונים (Collection) עם יכולת להוסיף ולמחוק איברים כאשר בכל פעם נרצה לשים נתונים אחרים
- נרצה מחלקה Point המייצגת נקודה במישור, כאשר לעיתים נרצה שהקורדינאטות (x,y) של הנקודה יהיו מסוג int ולעיתים נרצה שיהיו מסוג double

בעזרת הידע שיש לנו עד עתה נצטרך לבנות את המחלקות מספר פעמים, פעם אחת עבור כל טיפוס. אפשרות אחרת היא להשתמש בטיפוס object אך אפשרות זו לא מספיק טובה כי היא מחייבת אותנו לבצע casting בכל פעם שנרצה לעבוד עם המשתנה ובנוסף אנו עלולים לטעות ולהכניס למשתנה אובייקט מטיפוס שלא התכוונו אליו והקומפיילר לא יתריע על כך, אלא נקבל את השגיאה רק בזמן ריצה ב - casting הלא חוקי. השימוש ב box\unbox גוזל זמן ויוצר הרבה טעויות זמן ריצה ראה דוגמה

[01 ObjectStack]

Boxing – המרה מטיפוס value לטיפוס ref

```
string name = "Joe";

int age = 19;

object o = age; // boxing occurs here

// implicit boxing also occurs in the line below

Console.WriteLine( "{0} is {1}", name, age );
```



Unboxing – ניתן להמיר את הטיפוס חזרה ל value ע"י casting. שימו לב שהטיפוס חייב להיות בדיוק הטיפוס שממנו עשינו את ה boxing

```
class DynamicList {  
    public object GetItem( int index ) {  
        return ...;  
    }  
}  
...  
myList.Add(7);  
long x = (long)      myList.GetItem(0); //ERROR!!!  
long y =      (int) myList.GetItem(0);  
long z = (long) (int) myList.GetItem(0);
```

GENERIC

Generic זוהי מחלקה שאנו בונים המכילה משתנה מטיפוס כללי. ניתן להגדיר גם את סוג המשתנה כפרמטר, כך שנוכל ביצירת האובייקט לקבוע מה יהיה הטיפוס עבור האובייקט המסוים ובאובייקט אחר נקבע טיפוס אחר. בשיטה זו לא נצטרך לבצע casting מפני שאנו מגדירים את הטיפוס ואם ננסה להכניס ערך מטיפוס לא נכון בטעות נקבל שגיאת קומפילציה.

בכדי להשתמש ב - Generics יש להגדיר את הטיפוס כפרמטר באמצעות סוגריים משולשים <> מיד לאחר שם המחלקה לדוגמה:

```
class Point <T>  
{  
...  
}
```

```
}
```

לאחר מכן נוכל להשתמש בפרמטר זה בתוך המחלקה:

```
class Point <T>
{
    private T x;
    private T y;

    public T X
    {
        get{ return x; }
        set{x = value; }
    }

    public T Y
    {
        get {return y; }
        set {y = value; }
    }

    public Point()
    {
    }

    public Point(T x, T y)
    {
        X = x;
        Y = y;
    }

    public override string ToString()
    {
        return string.Format("X = {0}, Y = {1}", X, Y);
    }
}
```

כאשר ניצור את האובייקט נספק לו את הפרמטר של הטיפוס גם כן בסוגריים משולשים. לדוגמה Point כאשר ה - T הוא מסוג int:

```
Point<int> pInt = new Point<int>(10,5);
```

שימו לב שבדוגמה הבאה 2 השורות האחרונות יגררו שגיאת קומפילציה מפני שטיפוס הנתונים מוגדר מראש ולא ניתן להכניס ערך מסוג אחר:

```
static void Main(string[] args)
{
    Point<int> pInt = new Point<int>(10,5);
    Point<string> pString = new Point<string>("a", "b");
    pInt.X = "a";
    pString.X = 10;
}
```

עוד נקודות חשובות לגבי Generics:

- ניתן לקבל גם יותר מפרמטר אחד עבור מספר סוגים של משתנים
- ניתן לבנות כמעט כל אלמנט כ - generics כגון: class, method, delegate, interface
- ניתן לרשת מחלקה generics ולהגדיר לה את הטיפוסים כקבועים
- ניתן להשתמש ב default(T) שיתן את הערך ברירת מחדל לאותו טיפוס
- ב IL קיים הטיפוס הגנרי (מסומן ע"י ' בקוד). רק בזמן ריצה ה JIT מחליף לטיפוס האמיתי - חיסכון במקום עבור גודל ה EXE. לא צריך לייצר סתם סוגים מיותרים
- ניתן להגביל את הטיפוס שנהיה מוכנים לקבל כפרמטר באמצעות המילה השמורה where זה נקרא constraints שיכול להיות מסוג מחלקה, מבנה (struct), ממשק או ctor.

[02 My GenericStack]

דוגמה להורשת מחלקה generics:

```
class PointDouble : Point<double>
{
}
}
```

דוגמה להגבלת טיפוסים (נחייב את הטיפוס לממש את IComparable):

```
class Point <T>
    where T : IComparable
{
...
}
```

בספריות של .NET ישנם הרבה טיפוסים מוכנים המממשים את הרעיון של Generics וזמינים לשימושינו, לדוגמה:

- IComparable<T> - למימוש IComparable עם הפרמטר המתאים לפונקציה CompareTo
- List<T> - דומה ל ArrayList אך עם הגדרת הטיפוס של הנתונים ביצירת האובייקט

- Dictionary<TKey, TValue> - דומה ל Hashtable עם הגדרת הסוג של ה key ושל ה value ביצירת האובייקט

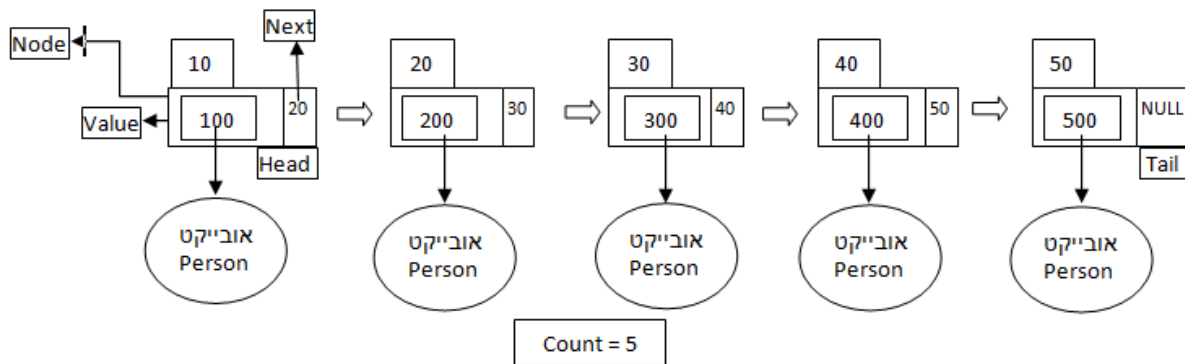
עוד דוגמה:

```
MyCollection<int> arr = new MyCollection<int>();  
class MyCollection<T>  
{  
    private T[] items = new T[100];  
    int index = 0;  
    public void Add(T val)  
    {  
        items[index++] = val;  
    }  
    public T GetVal(int index)  
    {  
        return items[index];  
    }  
}
```

ישנם מספר אוספים גנריים בשפה למשל:
[03 List, Dictionary, SortedDictionary]

LINKEDLIST – רשימה מקושרת

זהו "מבנה נתונים" (צורה בזכרון שמכילה נתונים בפורמט מסויים לדוגמה: stack, queue, array, arraylist) הבנוי בצורה הבאה:



יתרונות:

- יכולת להכניס ערך באמצע ולכן להכניס ערכים כך שהאוסף ימשיך להיות ממורכז
- יכולת למחוק ערך מהאמצע כל שלא ישאר "חור"

חסרונות:

- אין גישה ישירה לערך אלא יש צורך לרוץ ולחפש את הערך
- כל תא מכיל גם מצביע next ולעתים previous – בזבוז של 2 ייחוסים (8 בתים)

הרצה על האוסף והדפסה של כל ערך באוסף:

```
LinkedList<string> ll = new LinkedList<string>();
for (LinkedListNode<string> temp = ll.First; temp != null; temp = temp.Next)
    Console.WriteLine(temp.Value);
```

SORTEDLIST

שימוש:

```
SortedList<int, Person> arr = new SortedList<int, Person>();
arr.Add(p.ID, p);
```

אין להכניס את ערך ה-key פעמיים.

עוד אוספים :

Old	New
ArrayList	List<T>
Hashtable	Dictionary<K,V>
SortedList	SortedDictionary<K,V>
Stack	Stack<T>
Queue	Queue<T>

ממשקים גנריים:

Old	New
IList	IList<T>
IDictionary	IDictionary<K,V>
ICollection	ICollection<T>
IComparable	IComparable<T>
IComparer	IComparer<T>
IEnumerable	IEnumerable<T>
IEnumerator	IEnumerator<T>

[04 Queue, GenericQueue]

תרגיל




בנה מחלקה המדמה את List<T> וממשת מבנה של מערך דינאמי. יש להחזיק בתוך המחלקה מערך רגיל (לא Collection) ולטפל בכל מקרי הקצה. להלן תרשים המחלקה:

MyList<T>



Generic Class
















Fields

-  BUFFER_SIZE
-  count
-  innerList

Properties

-  Count
-  this

Methods

-  Add
-  AddRange
-  AllocList (+ 1 overload)
-  Clear
-  Contains
-  IndexOf
-  Init
-  Insert
-  MyList
-  Remove
-  RemoveAt
-  Sort
-  ToArray