בס"ד

Generics מאפשרת לכתוב מחלקות שיש בהם שימוש חוזר בסוגים שונים של משתנים, בלי להשתמש בהמרות, בלי Boxing ו Unboxing ועם Type safety.
ב Generics אנחנו יוצרים מעין תבנית למתודה או מחלקה.
בהגדרת המחלקה הגנרית נעביר עם תחביר מיוחד - בתוך סוגריים משולשות <> את הסוג של הפרמטר, וכך בהמשך נוכל לממש וליצור מופעים של המחלקה או להשתמש במתודה עם סוגים שונים של משתנים.
נראה דוגמה:

```
// Declare the generic class.
public class GenericList<T>
{
    public void Add(T input) { }
}
class TestGenericList
{
    private class ExampleClass { }
    static void Main()
    {
        // Declare a list of type int.
        GenericList<int> list1 = new GenericList<int>();
        list1.Add(1);

        // Declare a list of type string.
        GenericList<string> list2 = new GenericList<string>();
        list2.Add("");

        // Declare a list of type ExampleClass.
        GenericList<ExampleClass> list3 = new GenericList<ExampleClass>();
        list3.Add(new ExampleClass());
    }
}
```

כאשר יממשו את המחלקה החדשה שיצרנו, הקומפיילר יבדוק איזה Type העברנו, ויבנה מחלקה זהה, כאשר הוא מחליף כל מקום שכתבנו T לסוג שמימשנו.
בדוגמה הקודמת, הקומפיילר יצור 3 סוגים של מחלקות ויעשה בהם שימוש כל אחד במקומו.
הקומפיילר יצור מחלקה GenericLis~Int, GenericList~String ו GenericList~ExampleClass.

הקומפיילר יצר מחלקה שנראית כך:

```
public class GenericList~Int
{
    public void Add(int input) { }
}
```

בדוגמה הנ"ל: מיד אחר כך הקומפיילר יצור **מופע** של GenericLis~Int ויצביע עליו עם המשתנה שנקרא list1.
אם ננסה לכתוב `list1.Add("");` נקבל שגיאת קומפילציה, כי המתודה Add של GenericList~Int כמובן מצפה לקבל int.

הערה למתקדמים (זה בסדר לדלג! ומומלץ לפעמים!!): אם נפתח את קוד ה IL לא נראה את המימושים השונים של המחלקה הגנרית, נראה רק קריאות לקוד Native שגורמים לכך שבזמן ריצה, כלי הקימפול המיידי שמשמש את ה Framework (JIT Compiler), כלומר, הכלי שמתרגם מ IL לשפת המכונה, ייצור בזמן ריצה את המחלקות הנ"ל. את ה TypeSafe נקבל כמובן, כבר בזמן הקומפילציה.

מאחורי הקלעים, אכן יווצר בזיכרון מחלקה נוספת עבור כל סוג שמומש, רק שזה קורה בזמן ריצה.

המפתח להבנה טובה של Generics ב #C בניגוד למה שקורה בשפות אחרות,היא לחשוב על זה כ Type שנוצר בזמן קומפילציה, בפועל כאמור, זה יקרה בזמן ריצה. (ב Java למשל ישנה רק מחלקה אחת, והקומפיילר עורך עבורנו המרות ל Type המתאים עבור כל סוג שמימשנו).

השמות שהצגתי כשמות המחלקות שייוצרו, הם לא השמות המדויקים, ונועדו להמחשת העניין בלבד!

מחלקות ושיטות גנריות משלבות שימוש חוזר, בטיחות סוג (type safety) ויעילות באופן שמקבילותיו הלא גנריות לא יכולות .

לרוב, משתמשים ב Generics עם אוספים ומתודות שפועלות עליהם .

אם כך, Generics אפשרה לנו לעצב מחלקות ומתודות שדוחות את הגדרת ה Type של הפרמטר עד שישתמשו בו ויממשו אותו בקוד, כלומר עד שיצרו מופע שלו.

יש המון מחלקות גנריות שמובנות בדוטנט.

ה namespace שנקרא System.Collections.Generic מכיל מספר מחלקות גנריות שמטפלות באוספים. עוד נעשה בהן שימוש רב.

נ.ב. אם תמצא אוספים לא גנריים, כגון ArrayList דע כי הן אינן מומלצות לשימוש כלל, ומתוחזקים לצורכי תאימות עם גרסאות ישנות של דוטנט.

כמו בדוגמה למעלה, תוכל גם אתה ליצור סוגים ושיטות כלליות בהתאמה אישית כדי לספק פתרונות כלליים ודפוסי עיצוב משלך, בטוחים ויעילים.

לטיפול באוספים, ברוב המקרים, עליך להשתמש במחלקת List <T> המסופקת על ידי NET . במקום ליצור משלך (למרות שזה מגניב...)

נראה דוגמה נוספת למחלקה גנרית. שימו לב שפרמטר T משמש במספר מיקומים שבהם בדרך כלל יש להגדיר במפורש את הסוג. הוא משמש מזהה לסוג של ערך מוחזר, מזהה לסוג

של פרמטר של מתודה, וגם משמש כסוג של שדה. נראה את זה בדוגמה הבאה, שמממשת רשימה מקושרת פשוטה.

```
// type parameter T in angle brackets
public class GenericList<T>
{
    // The nested class is also generic on T.
    private class Node
    {
        // T used in non-generic constructor.
        public Node(T t)
        {
            next = null;
            data = t;
        }

        private Node next;
        public Node Next
        {
            get { return next; }
            set { next = value; }
        }

        // T as private member data type.
        private T data;

        // T as return type of property.
        public T Data
        {
            get { return data; }
            set { data = value; }
        }
    }

    private Node head;

    // constructor
    public GenericList()
    {
        head = null;
    }

    // T as method parameter type:
    public void AddHead(T t)
    {
        Node n = new Node(t);
        n.Next = head;
        head = n;
    }

    public IEnumerator<T> GetEnumerator()
    {
        Node current = head;

        while (current != null)
```

```
        {
            yield return current.Data;
            current = current.Next;
        }
    }
}
```

T מופיע בדוגמה למעלה

- כסוג של פרמטר שיטה בשיטת AddHead.
- כסוג ההחזרה של מאפיין הנתונים במחלקת Node.
- כסוג ה private member במחלקה המקוננת.

לא רק מחלקות

לא כל המחלקה חייבת להיות גנרית! ניתן להגדיר במחלקה רגילה, מתודות גנריות (וגם לפרמטרים של מתודות במחלקה "רגילה" – ללא Generics, ניתן להגדיר אילוצים (נושא האילוצים בהמשך). לדוגמה:

```
class PlayGround
    {
        void Play()
        {
            GenericList<int> intList = new GenericList<int>();
            intList.Add(68);
            /*ERORRRRRRR*/
            //intList.Add("Ata Retzini?");

            GenericList<string> stringList = new GenericList<string>();
            stringList.Add("Oved!");
            /*ERORRRRRRR*/
            //stringList.Add(68);

        }

        public static void PrintXandY<T>(T x, T y)
        {
            Debug.WriteLine("First variable = {0}, Second variable = {1}", x, y);
        }

}
```

כמו"כ, ניתן להגדיר ממשקים (interface) גנריים.

אין צורך לאמץ את המוח להבין את המשמעות של זה, נשתמש המון בכאלו שמובנים בדוטנט, בהמשך הקורס.

הגדרה מפורשת ונסתרת

לפעמים, במתודות ניתן להשתמש ב generics בצורה מפורשת, והקומפיילר מאפשר לנו להשתמש בתחביר שאינו מפורש, ב VS מתקדם גם נקבל הודעת אזהרה שניתן לפשט את התחביר ולא להעביר את הארגומנט שמציין את הסוג.

ראו דוגמה להבנה, ושימו לב כי שתי צורות השימוש במתודה הגנרית זהות, הקומפיילר מבין שמימשנו StamMehod של int אפילו שלא ציינו את זה במפורש :

```
static void StamMethod<T>(T t) { }

        public static void main()
        {
            StamMethod<int>(5);
            /*The above Equals to: */
            StamMethod(5);

        }
```

סיכום

- השתמש ב Generics על מנת למקסם את השימוש החוזר בקוד, בטיחות הקוד והביצועים.
- השימוש הנפוץ ביותר בגנריקה הוא יצירת מחלקות לטיפול באוספים.
- ספריות מחלקות NET. מכילה מספר כיתות אוסף גנריות במרחב השמות System.Collections.Generic. יש להשתמש באוספים הגנריים בכל מקום אפשרי במקום בישנות כגון ArrayList שנמצא במרחב השמות System.Collections.
- תוכל ליצור ממשקים, מחלקות, שיטות, אירועים ו members עם גנריקה משלך.
- מחלקות גנריות עשויות להיות מוגבלות כדי לאפשר גישה לשיטות על סוגי נתונים מסוימים.
- ניתן לקבל מידע על הסוגים המשמשים בסוג נתונים גנרי בזמן ריצה באמצעות Reflection.

Constraints – הגבלות

אילוצים מציינים את הציפיות שלנו מה Type parameter. הצהרה על אילוץ מוודאת שישתמשו במתודות ובמחלקות הגנריות שכתבת רק באופן שתכנת, ומאפשרת להשתמש במתודות ומאפיינים ששייכים לסוג שקיבלנו כפרמטר בתוך המתודה או בתוך המחלקה.
כמו תמיד, בכדי להבין נעבור מיד לדוגמה.

ניתן לשלב יותר מאילוץ אחד על כל סוג.

לטובת Reference עברי לשימוש עתידי, אני מעתיק ומתרגם את הטבלה המקורית של מייקרוסופט על כל האילוצים החדשים שהגיעו בגרסאות האחרונות של דוטנט, אבל, ממליץ

לכם מאוד לא להתעמק בהם ולנסות להבין את הצורך של כל אחד, זה לא יתרום לכם מאומה כרגע ורק יבלבל.

תקראו רק את המרכזיים שבהם שהודגמו גם בכיתה!

Constraint	Description
`where T : struct`	הארגומנט שנעביר לפרמטר ה"סוג" יהיה non-nullable value type (למידע נוסף בנושא Nullable value types). מאחר ולכל ה Value types יש בנאי ברירת מחדל שלא מקבל פרמטרים, אילוץ זה כול את האילוץ ()new ולא ניתן להוסיף ולשלב את אילוץ ה ()new יחד עם אילוץ זה. כמו"כ, הוא לא משתלב עם אילוץ ה unmanaged.
`where T : class`	ארגומנט הסוג יהיה reference type. האילוץ חל ומאפשר גם כל מחלקה, ממשק, Delegate, או מערך. ב C# 8 ומעלה, האילוץ יחייב אותנו לממש את T עם סוג שלא אמור להצביע על null (למשל: Person ולא Person?).
`where T : class?`	כמו הקודם, רק אפשרי nullable types (Person?).
`where T : notnull`	Valur or reference type ובלבד שאינו מוגדר עם סימן שאלה.
`where T : default`	נועד למקרה מאוד מסוים בו יורשים מחלקה (או מתודה) גנרית שאין בה שום אילוץ, ורוצים לאפשר nullable ממליץ לא להתעמק בזה, עד שאולי אם אי פעם תזדקקו לזה.
`where T : unmanaged`	מאלץ להשתמש בסוג implies non-nullable unmanaged type the `struct` constraint לא בא לו טוב עם האילוצים ()new או struct
`where T : new()`	שימושי כאשר רוצים לאתחל מופעים של מחלקה בתוך הקוד של המחלקה הגנרית. מאלץ אותנו להעביר בארגומנט הסוג, מחלקה שיש לה בנאי ללא פרמטרים. אם משתמשים באילוץ זה עם אילוצים נוספים, אילוץ זה צריך להיות מוגדר אחרון ברשימת האילוצים.
`where T :` *<base class name>*	אילוץ נפוץ ושימושי ביותר, שמגביל אותנו למחלקה או היורשים ממנה. לא מאפשר Person? (אחרי גרסה 8 של C#).
`where T :` *<base class name>?*	כנ"ל עם מחלקות Nullable
`where T :` *<interface name>*	נפוץ מאוד ושימושי ביותר, להכריח אותנו לקבל ב T ממשק, או מחלקה שמממשת את הממשק. ניתן להגדיר באילוץ שני ממשקים (כלומר, הסוג יממש את הממשק האחד **או** את השני).
`where T :` *<interface name>?*	כנ"ל, אך מאפשר גם Nullable (שמממש כמובן את הממשק שכתבנו את שמו ב where.
`where T : U`	אם יש לי שני פרמטרים של סוג, אני יכול לגדיר שהפרמטר השני הוא אותו סוג של הראשון, או מחלקה אחרת שיורשת מהראשון.

Constraint	Description
	למשל אם הגדרתי Func<U,T> () where U : Person where T : U

הגבלות נוספות

ב Interfaces וב delegates (שעוד לא למדנו) ניתן להגדיר גם שהפרמטר "סוג" שנקבל, ישמש במימוש רק כערך שנכנס לפונקציה כפרמטר (או set של Property). לחילופין, ניתן להגדיר שהפרמטר ישמש רק כערך מוחזר של פונקציה (או ב get של Property).

לדוגמה:

```
interface ICoManageZoo<out T>
    {
        //void Add(T t); /*Compiler ERRORRRR*/
        T GetT(string name);
    }
    interface IContraManageZoo<in T>
    {
        void Add(T t);

        //T Get(string name); /*Compiler ERRORRRR*/

    }
```

Variant

ההגבלות הקודמות in & out, מסייעות לנו ליצור **מעין** ירושות של שימוש ב Interfaces גנריים.

שימו לב שלמרות ש Student יורש מ Person, List<Student> אינו ממש בירושה את IEnumerable<Person> (למרות שהוא כן ממש את IEnumerable<T>.

למרות זאת, הקוד הבא תקף:

```
IEnumerable<Person> people = new List<Student>() { Student1, Student2 };
```

הנושא קצת קשה לתפיסה בשלב הזה, מספיק שנדע שהוא קיים. לא ממליץ להתעמק בו בכלל בשבלב הזה שלכם, וההיפך!! אני ממליץ מאוד **שלא ללמוד אותו כעת** (בדוגמאות הקוד שצורפו לשיעור, יש דוגמה, רק לטובת צורך עתידי אם אי פעם יהיה לכם, ותזכרו שיש דוגמה כזו 😊)!

תרגילי כיתה ובית

1. כתבו פונקציה Combine שמחברת מערכים שהתקבלו כפרמטר מכל סוג.
2. ממשו אוסף Read only גנרי שמבוסס על מערך, למחלקה יהיה מתג (סוויטץ) – משתנה, שיגדיר אם ניתן כרגע להוסיף איברים למערך. זרוק InvalidOperationException אם ניסו להוסיף ערך בזמן ש IsReadOnly = true.
3. צרו אוסף שבו האיברים תמיד ממוינים, האוסף יעבוד על char, int, double לא על Classes.
4. צרו מחלקה גנרית שיודעת ליצור מופעים חדשים של מחלקת ה T שקיבלנו כ Type פרמטר.
   ל T יהיה מאפיין בשם ID הקוד יאתחל את המאפיין.
5. כתבו Stack גנרי (מתודות: Push, Pop, Peek)
6. כתבו Queue גנרי (מתודות: Enqueue, Dequeue)
7. ממשו מחלקה שמדפיסה את הסוג של המחלקה, ואת המאפיין Name שבמחלקה. איך נוודא שיש מאפיין בשם Name במחלקה?

התקדמנו, מה עכשיו?

אז כעת, כשאנחנו מבינים את נושא הגנריקה, ואפילו איך ליישם אותו בעצמנו, בואו נזכור שבדרך כלל לא נרצה ומיותר לנו להמציא מחדש את הגלגל.
לטובת אוספים, רשימה מקושרת עם יכולות תחביר של מערך, בדרך כלל נשתמש בבאופים המובנים ב System.Collection.Generics.

IList<T>, List<T>, Dictionary<TKey,Tvalue>

```
List<int> intList = new List<int>();
IList<int> intList2 = new List<int>();

var stringList=new List<string>();
stringList.Add("Some string");
```

תרגילי כיתה ובית

כרגע אנחנו צריכים המון מיומנות, והיכולות שלנו השתדרגו מאוד.
נשחזר את כל שיעורי הבית של Indexers ו Generics עם Lists, כל מה שרלוונטי.

המרצה יעביר את התרגילים בעל פה בשלב זה (לא הספקתי ☹), גרסת המסמך הבאה תכלול את התרגילים במפורש, אבל לא בטוח שזה יקרה במחזור הנוכחי
