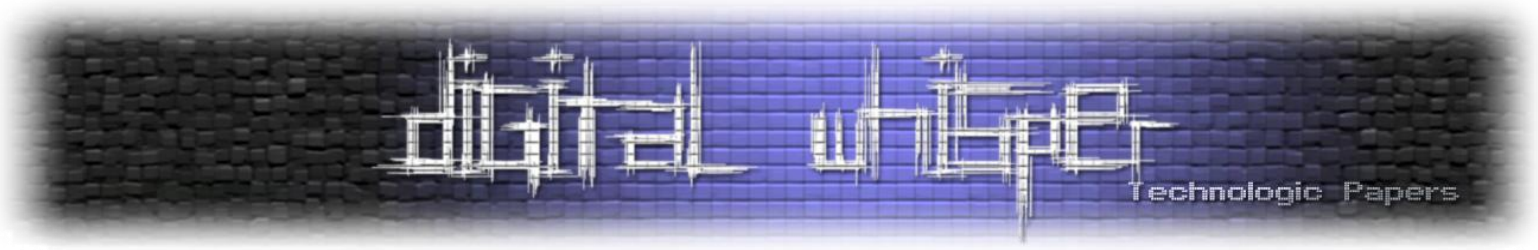

תכנות גנרי בשפת #C

מאת ניר אדר (UnderWarrior)

דוגמת פתיחה ומוטיבציה

תכנות גנרי הופיע לאורך השנים בשפות מונחות עצמים רבות. הרעיון הוא מימוש אלגוריתמים שאינם תלויים בטיפוס הנתונים עליהם הם פועלים.

באופן רגיל, כאשר אנחנו כותבים קוד בשפה Strongly Typed (כדוגמאת C#, Java), אנחנו מגדירים בצורה מפורשת את הטיפוס עליו אנו עובדים, int למשל, ויוצרים קוד שמתאים לסוג טיפוס זה. לעומת זאת, כשאנחנו כותבים אלגוריתמים כלליים, אנחנו מסוגלים לתאר את הפעולות מבלי להזדקק לציון הסוג.

דוגמא קלאסית למקרה זה היא הפונקציה swap. פונקציה זו מקבלת שני משתנים מסוג int ומחליפה בין ערכיהם. נדגים את הפונקציה ופונקצית main המשתמשת בה:

```
static void swap(ref int i, ref int j)
{
      int temp = i;
      i = j;
      j = temp;
}

static void Main(string[] args)
{
      int i = 5, j = 7;
      Console.WriteLine("i = " + i.ToString() + ", j = " +
j.ToString());
      swap(ref i, ref j);
      Console.WriteLine("i = " + i.ToString() + ", j = " +
j.ToString());
}
```

הפלט הוא:

```
i = 5, j = 7
i = 7, j = 5
```

swap מחליפה בין ערכי המשתנים שהיא מקבלת על ידי שמירת ערכו של אחד מהם במשתנה זמני ולאחר מכן מעבירה את הערך שהיה במשתנה אחד אל המשתנה השני, והערך שהיה בשני אל המשתנה הראשון.

איך תראה הפונקציה אם נרצה להתאימה למשתנים מסוג double? זהה לחלוטין! רק שנצטרך לכתוב double בכל מקום שבו כרגע כתוב int:

```
static void swap(ref double i, ref double j)
{
      double temp = i;
      i = j;
      j = temp;
}
```

דבר זה נכון למעשה עבור כל סוג של משתנים שנרצה לכתוב עבורם פונקצית swap – הגוף יראה זהה ורק סוג המשתנה ישתנה. אנחנו רוצים למצוא דרך יעילה לכתוב זאת בלי לשכפל את הקוד שלנו פעמים רבות. (משפט ידוע המלווה שפות רבות אומר כי קוד המופיע פעמיים – מופיע פעם אחת יותר מדי).

הפתרון הינו תכנות גנרי, תכנות שבו איננו מגדירים את סוג המשתנה אלא מציינים "סוג כלשהו", הקוד בשפת C# יראה כך:

```
static void swap<T>(ref T i, ref T j)
{
      T temp = i;
      i = j;
      j = temp;
}
```

אנחנו בעצם יוצרים תבנית, וכשנקרא ל-swap השפה תזהה את טיפוס המשתנים שלנו ותיצור פונקציית swap המתאימה לטיפול בערכים אלה. הגנריות מבחינת תחביר מסומנת על ידי <T> - אנחנו מציינים ש-T הוא סוג כללי כלשהו.

אם נחליף את פונקצית ה-swap בדוגמת הפתיחה בפונקציה זו התוכנית תעבוד ללא שינוי. בנוסף, אם פתאום נצטרך פונקצית swap בין double, לא נצטרך לכתוב פונקציה מיוחדת באופן מפורש – C# תדאג לתבנית מתאימה גם עבור מקרה זה.

מקום קלאסי נוסף בו משתמשים בתכנות גנרי הוא Collections – רשימות, עצים ושאר מבני הנתונים – הרי מבנה הנתונים (והפעולות עליו) נשארים זהים תמיד. סוגי הנתונים הנשמרים בכל פעם משתנים ולכן מימוש של מבני נתונים הוא דוגמא שימושית וחשובה לצורך שלנו בתכנות גנרי.

קצת היסטוריה

תכנות גנרי קיים בצורות שונות עוד מ-1970. בהמשך, שפת C++ הציגה תכנות גנרי בצורת תבניות (templates) – מבנה שמבחינת תחביר זהה מאוד לפתרון לתכנות גנרי של C# המכונה generics.

בשפת C# במקור לא היתה תמיכה בתכנות גנרי, מכיוון שכל המחלקות נגזרות מהאובייקט object, היה ניתן ליצור לצורך העניין רשימה כללית שתכיל אובייקטים מכל סוג שהוא ללא צורך בתבניות. עם זאת, ל-generics יש מספר יתרונות שגרמו למעצבי השפה להכניס אותם לשפה החל מ-Net 2005.

- **Type Safety -** במקרים רבים נרצה ליצור רשימה שיש לה type checking, שנקבל הודעת שגיאה בזמן קומפילציה במידה ומנסים להכניס אליה אובייקט מסוג לא נכון, ושנדע שאנחנו יכולים להשוות בין כל האובייקטים השונים הנמצאים באותה הרשימה. לדוגמא, ללא תכנות גנרי הקוד הבא היה עשוי לגרום לשגיאה בזמן ריצה:

```
ArrayList intList = new ArrayList();
// some ops on the list here
// ...
foreach (int x in intList)
{
      // some comments here
}
```

במקרה ובו לא כל האיברים ברשימה יהיו int, התוכנית תקרוס. אנחנו נהיה מעוניינים למנוע מצב כזה ולדאוג שאם ננסה להכניס איבר שאינו int נקבל שגיאה עוד בזמן הקומפילציה.

- **ביצועים -** חיסכון בפעולות Boxing/Unboxing. כל פעם שאנחנו שומרים מחלקה/ערך בתוך מבנה נתונים כללי – הוא עובר פעולת boxing, וכשמוציאים אותו הוא עובר פעולת unboxing. אלו פעולות יקרות שפוגעות משמעותית ביעילות מבנה הנתונים. שימוש ברשימה המיועדת ל-int בלבד, למשל, יכול למנוע את ביצוע פעולות אלו ולשפר משמעותית את ביצועי התוכנית.

- **Code reuse -** כדי להשיג פונקציונליות של type safety ללא Generics אנחנו צריכים מבנה נתונים נפרד לכל סוג, למשל רשימה מקושרת של int, רשימה מקושרת של char וכדומה. ראינו למשל בדוגמאת swap שללא generics היינו צריכים לכתוב פונקציה נפרדת עבור int, פונקציה אחרת ל-double, וכו'. Generics מאפשר לנו למנוע את שכפול הקוד ולכתוב את הפונקציה פעם אחת. נושא זה חשוב במיוחד כשאנחנו כותבים פונקציה מעט מורכבת יותר. אם הפונקציה משוכפלת, לעדכן את כל העותקים שלה יכול להיות סיוט ופתח משמעותי לשגיאות.

השם generics בא לציין שאנחנו מגדירים מחלקות ומתודות גנריות. (בשפות אחרות השם היה templates, תבניות, ובא להגיד שיוצרים תבנית, ולא פונקציה "אמיתית". שפת C# בחרה להדגיש בשם את הרעיון שאנחנו באים לתכנת משהו כללי שהוא לא ספציפי לסוג מסוים.

כמו שכבר ציינתי מקודם, השימוש העיקרי שעושים ב-generics בשפה הוא ב-collections. C# 2.0 הציגה ספריית מחלקות חדשה, תחת המסלול System.Collections.Generic הממומשות באמצעות generics להשגת היתרונות שמנינו. **מומלץ להשתמש במחלקה זו בכל מקום אפשרי במקום הספריות הישנות מתוך System.Collections.**

מתודות גנריות

ב-C# ניתן להגדיר מתודות גנריות. דוגמא למתודה גנרית ולשימוש בה נראית כך:

```
using System;

class GenericMethod
{
	static void Main(string[] args)
	{
		// create arrays of int and double
		int[] intArray = { 1, 2, 3, 4, 5, 6 };
		double[] doubleArray =
			{ 1.1, 2.2, 3.3, 4.4, 5.5, 6.6, 7.7 };

		// pass an int array argument
		PrintArray(intArray);

		// pass a double array argument
		PrintArray(doubleArray);
	}

	// output array of all types
	static void PrintArray<E>( E[] inputArray )
	{
		foreach (E element in inputArray)
			Console.Write( element + " " );
		Console.WriteLine( "\n" );
	}
}
```

הפונקציה PrintArray<E> היא פונקציה גנרית (סוג המשתנה מסומן כ-E). בכל פעם הפונקציה מדפיסה מערך מסוג אחר.

נביט בדוגמא ונציג מספר נקודות חשובות בה:

1. **הקריאה לפונקציה הגנרית:** בניגוד ל-++C, הקריאה לפונקציה נעשית על ידי שמה, בלי שצריך לציין באופן מפורש מהו E בקריאה לפונקציה, כלומר נכתוב PrintArray(intArray) והשפה תנחש לבד את הסוג בו משתמשים. אם נרצה, ניתן גם לציין את הסוג במפורש, למשל לכתוב PrintArray<int>(intArray) על מנת להגיד לשפה שאנחנו מעוניינים להשתמש בגרסת הפונקציה עם הסוג int. יש צורך לציין במפורש את הסוג גם במקרים בהם #C לא תצליח לנחש אותו לבד, ותשלח הודעת שגיאה

2. **לא כל המחלקה חייבת להיות גנרית**: הבדל נוסף משפת ++C הוא שלא כל המחלקה חייבת להיות גנרית כדי שנוכל להשתמש בה במתודות גנריות – מתודה ספציפית במחלקה יכולה להיות גנרית בזמן שהמחלקה הינה מחלקה רגילה.

3. **הפרמטר יכול להיות מיוצג על ידי כל מזהה**, ולאו דווקא T או E. מקובל שהפרמטר הוא באותיות גדולות ולעתים מקובל להשתמש באותיות מסויימות (T עבור Type, K עבור Key וכדו')

מחלקה גנרית

מחלקה שלמה יכולה להיות גנרית, כלומר לקבל סוג גנרי שעובר לאורך כל המחלקה. הדוגמא הקלאסית היא דוגמת ה-Collections, למשל יצירת רשימה כללית ולאחר מכן יצירת רשימה של int. באמצעותה.

מבחינת התחביר יש לחלק את התחביר לשני חלקים- התחביר הנדרש לצורך כתיבת מחלקה גנרית, והתחביר הנדרש לצורך שימוש במחלקה הגנרית שהגדרנו.

מבחינת התחביר ליצירת מחלקה גנרית:

- הפרמטר T מופיע בסוגריים משולשות שם המחלקה. בדומה לשפת ++C, T הוא אותו T בכל חלקי המחלקה.
- בשאר המתודות של המחלקה לא צריך לשים סוגריים משולשות לציין שמדובר בתבנית – כל הפונקציות הן גנריות בהינתן שהן חלק ממחלקה גנרית.

מבחינת התחביר לשימוש במחלקה הגנרית:

- כאשר יוצרים מופע של המחלקה, עלינו לציין כחלק משם המחלקה מהו הסוג שאנחנו יוצרים.

הקוד הבא מציג רשימה גנרית בסיסית ומדגים את התחביר. הסוג של הנתונים הינו T, סוג כלשהו:

```
// 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;
                }
        }
}
```

שימוש לדוגמא במחלקה:

```
class TestGenericList
{
        static void Main()
        {
                // int is the type argument
                GenericList<int> list = new GenericList<int>();

                for (int x = 0; x < 10; x++)
                {
                        list.AddHead(x);
                }

                foreach (int i in list)
                {
                        System.Console.Write(i + " ");
                }
                System.Console.WriteLine("\nDone");
        }
}
```

הגדרנו כי GenericList היא מחלקה גנרית עם פרמטר T, שהוא סוג הנתונים הנשמר במחלקה. הגדרנו מחלקה פנימית Node לשמירת צומת ברשימה, בתור חלק ממחלקה גנרית, המחלקה Node גנרית באופן אוטומטי.

מגבלות על הפרמטרים

שפת #C מאפשרת לנו להגדיר מגבלות על הפרמטרים הגנרים אותם נקבל (T לא יהיה מסוג כלשהו, אלא יהיה חייב לעמוד במגבלות שנדרוש).

המגבלות ש-#C מאפשרת לנו להגדיר (לקוח מהגדרת השפה ב-MSDN):

Constraint	Description
where T: struct	The type argument must be a value type. Any value type except Nullable can be specified. See Using Nullable Types (C# Programming Guide) for more information.
where T : class	The type argument must be a reference type, including any class, interface, delegate, or array type.
where T : new()	The type argument must have a public parameterless constructor. When used in conjunction with other constraints, the new() constraint must be specified last.
where T : <base class name>	The type argument must be or derive from the specified base class.
where T : <interface name>	The type argument must be or implement the specified interface. Multiple interface constraints can be specified. The constraining interface can also be generic.
where T : U	The type argument supplied for T must be or derive from the argument supplied for U. This is called a naked type constraint.

לדוגמא, נגדיר מחלקה המייצגת עובד ורשימה המטפלת בעובדים:

```
public class Employee
{
      private string name;
      private int id;

      public Employee(string s, int i)
      {
            name = s;
            id = i;
      }

      public string Name
      {
            get { return name; }
            set { name = value; }
      }

      public int ID
      {
            get { return id; }
            set { id = value; }
      }
}

public class GenericList<T> where T : Employee
{
      private class Node
      {
            private Node next;
            private T data;

            public Node(T t)
            {
                  next = null;
                  data = t;
            }

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

            public T Data
            {
                  get { return data; }
                  set { data = value; }
            }
      }
```

```
        private Node head;

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

        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;
                }
        }

        public T FindFirstOccurrence(string s)
        {
                Node current = head;
                T t = null;

                while (current != null)
                {
                        //The constraint enables access to the Name property.
                        if (current.Data.Name == s)
                        {
                                t = current.Data;
                                break;
                        }
                        else
                        {
                                current = current.Next;
                        }
                }
                return t;
        }
}
```

הגדרנו מחלקה בשם Employee, ואז הגדרנו מחלקה שהיא רשימה שמקבלת רק אובייקטים מסוג Employee או מחלקות הנורשות ממנה (את זאת דרשנו באמצעות ההגבלה על הסוג). נשים לב שברגע שהגדרנו הגבלה זו, אנחנו מסוגלים לגשת לשדות של Employee בתוך המתודות של המחלקה. ההגבלה מאפשרת לנו כוח ביטוי נוסף, מכיוון שיש לנו ידע נוסף על הסוג.

דוגמא לתחביר - מחלקה עם מספר פרמטרים גנריים ועם הגבלות שונות על כל פרמטר:

```
class SuperKeyType<K, V, U>
    where U : System.IComparable<U>
    where V : new()
{ }
```

ממשקים גנריים

גם ממשקים (interface) מסוגלים להיות גנריים. צורת הכתיבה שלהם דומה לצורת הכתיבה של מחלקות גנריות:

```
interface IDictionary<K, V>
{
}
```

ממשק גנרי המקבל שני טיפוסים כלשהם.

הממשקים Icomparable, Ienumerable קיבלו גרסאות גנריות גם הם, כדי למנוע פעולות של Boxing ו-Unboxing. כמו ב-Collections, גם בהם מומלץ להשתמש בכל מקום בו ניתן.

דוגמא נוספת למקרה בו כותבים איסורים מרובים ומשתמשים גם בממשקים הגנריים:

```
class Stack<T> where T : System.IComparable<T>, IEnumerable<T>
{
}
```

ערך ברירת מחדל

בעיה שעולה כשאנחנו רוצים לאתחל משתנה גנרי היא אתחול ערך "ברירת מחדל" לערך הגנרי. ערכי ברירת מחדל למשתנים רגילים מוגדרים ב-#C כך- אם המשתנה מסוג int, הערך הוא 0, אם המשתנה הוא מספר עשרוני הערך הוא 0.0, ואם המשתנה הוא מחלקה אזי ערך ברירת המחדל הוא null. (ערכים אלו מושמים באופן אוטומטי למשתנים אם אנחנו מגדירים אותם ללא קביעת ערך).

הבעיה בעת שימוש במשתנה generic נובעת מכך שאנחנו אפילו לא יודעים אם נקבל ערך שהוא by value או by reference.

הפתרון של שפת #C הוא שימוש במילת המפתח default במשמעות חדשה של החזרת ערך ברירת המחדל עבור הסוג המועבר:

```
// The following declaration initializes temp to
// the appropriate  default value for type T.
T temp = default(T);
```

סיכום

במאמר זה נגעתי על קצה המזלג בתכנות גנרי בשפת #C. הנקודה החשובה ביותר שרציתי להעביר במסמך זה היא קיומן של אפשרויות תכנות אלו והשימושיות שלהן. אם תרצו להשאר עם דגש אחד ממאמר זה, הדגש הוא להשתמש רק ב-collections הממומשים בעזרת תכנות גנרי – יעילות התוכניות שלכם תשתפר בצורה משמעותית ובאגים יחסכו מזמן הריצה.

תכנות גנרי יכול לפתור בצורה אלגנטית בעיות רבות של שכפול קוד ואני מקווה שאחרי מאמר זה תמצאו את ההזדמנות לשלב אותו בקוד שלכם.