القوائم المترابطة والأدوات العامة
Linked Lists and Generics

الكلمات المفتاحية

الصفّ مع مرجع لنفسه، القوائم المترابطة، الأعضاء العامة، الصفّ العام.

ملخص الفصل

يوضّح الفصل الحالي بعض المفاهيم المتعلقة ببنى المعطيات كالقوائم المترابطة. ويبيّن دور الأعضاء العامة والصفوف العامة في كتابة رمّاز مصدري بطريقة مختصرة وأكثر سهولة.

الأهداف التعليمية

يتعرّف الطالب في هذا الفصل على:

- كيفية إنشاء قائمة مترابطة
- أهمية الأعضاء العامة وكيفية استخدامها وتقييد أنماطها
- إنشاء صفّ عام
- استخدام الصفّ العام `LinkedList`

مقدمة

على الرّغم من الدور الكبير الذي تلعبه المصفوفة كبنية معطيات شائعة الاستخدام، إلّا أنّ تثبيت عدد عناصرها قد يعيق استخدامها في الحالات التي تطلب حجزاً ديناميكياً للذاكرة. وفي مثل هذه الحالات، يتمّ اللجوء إلى القوائم المترابطة التي تعتمد على مبدأ استخدام الصفّ كمرجع لنفسه.

1. الصفوف مع مرجع لنفسها

يمكن لصفّ أن يحتوي على عضو يعمل كمرجع لغرض من نفس نمط الصفّ. وفي هذه الحالة، نُسمّي الصفّ بالصفّ ذي المرجع لنفسه Self-Referential Class.

ويمكن استخدام الصفّ ذي المرجع لنفسه لإنشاء سلسلة مترابطة من أغراض هذا الصفّ تشكل بنى معطيات مفيدة مثل القائمة List والمكدّس Stack والرتل Queue والشجرة Tree. ويمثّل الشكل الآتي قائمة مؤلّفة من غرضين مرتبطين مع بعضهما:

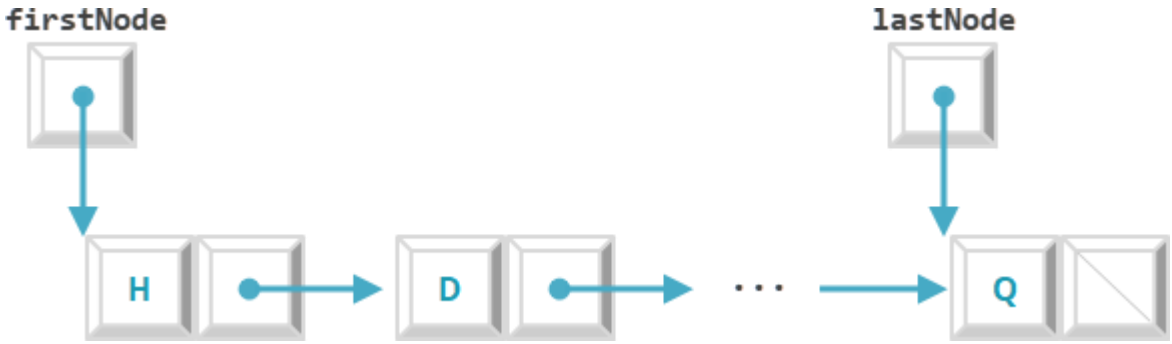
مثال:

يوضّح الرمّاز التالي مثالاً عن الصفّ ذي المرجع لنفسه Self-Referential Class.

```
class Node {
 public int Data { get; set; } // store integer data
public Node Next { get; set; } // store reference to next Node
 public Node(int dataValue)  {
    Data = dataValue;
 } // end constructor
} // end class Node
```

2. القوائم المترابطة

تتكوّن القائمة المترابطة من مجموعة من الأغراض المرتبطة مع بعضها بشكل تسلسلي، وندعو كلّ عنصر من هذه الأغراض بالعقدة Node. ويتمّ التعامل مع القائمة من خلال مرجع (مؤشِّر) للدلالة على أوّل عقدة فيها ويمكن استخدام مرجع (مؤشِّر) للدلالة على آخر عقدة أيضاً. وتتميّز العقدة الأخيرة من القائمة بوجود القيمة null في مرجعها للدلالة على انتهاء القائمة. ويوضّح الشكل الآتي مثالاً عن قائمة مترابطة:



مثال:

في الرمّاز التالي، نوضّح كيفية التعامل مع القوائم المترابطة، ثمّ نشرح أهم الطرائق المستخدمة للقيام بذلك:

```
// LinkedListLibrary.cs
// ListNode, List and EmptyListException class declarations.
using System;
namespace LinkedListLibrary
{
// class to represent one node in a list
// Self-referential Node class declaration.
class ListNode
{

// automatic read-only property Data
public object Data { get; private set; }

// automatic property Next
public ListNode Next { get; set; }

// constructor to create ListNode that refers to dataValue
// and is last node in list
public ListNode(object dataValue) : this(dataValue, null)
```

```
{
} // end constructor

// constructor to create ListNode that refers to dataValue
// and refers to next ListNode in List
public ListNode(object dataValue, ListNode nextNode)
{
Data = dataValue;
Next = nextNode;
} // end constructor
} // end class ListNode

// class List declaration
public class List
{
private ListNode firstNode;
private ListNode lastNode;
private string name; // string like "list" to display
// construct empty List with specified name
public List(string listName)
{
name = listName;
firstNode = lastNode = null;
} // end constructor

// construct empty List with "list" as its name
public List() : this("list")
{
} // end default constructor

// Insert object at front of List. If List is empty,
// firstNode and lastNode will refer to same object.
// Otherwise, firstNode refers to new node.
public void InsertAtFront(object insertItem)
{
if (IsEmpty()) firstNode = lastNode = new ListNode(insertItem);
else firstNode = new ListNode(insertItem, firstNode);
}// end method InsertAtFront
```

```
// Insert object at end of List. If List is empty,
// firstNode and lastNode will refer to same object.
// Otherwise, lastNode's Next property refers to new node.
public void InsertAtBack(object insertItem)
{
if (IsEmpty()) firstNode = lastNode = new ListNode(insertItem);
else lastNode = lastNode.Next = new ListNode(insertItem);
} // end method InsertAtBack

// remove first node from List
public object RemoveFromFront()
{
if (IsEmpty()) throw new EmptyListException(name);
object removeItem = firstNode.Data; // retrieve data
// reset firstNode and lastNode references
if (firstNode == lastNode) firstNode = lastNode = null;
else firstNode = firstNode.Next;
return removeItem; // return removed data
} // end method RemoveFromFront

// remove last node from List
public object RemoveFromBack()
{
if (IsEmpty()) throw new EmptyListException(name);
object removeItem = lastNode.Data; // retrieve data

// reset firstNode and lastNode references
if (firstNode == lastNode) firstNode = lastNode = null;
else
{
ListNode current = firstNode; // loop while current.Next is not lastNode
while (current.Next != lastNode) current = current.Next;
// move to next node // current is new lastNode
lastNode = current; current.Next = null;
} // end else
return removeItem;  // return removed data
} // end method RemoveFromBack
// return true if List is empty
```

```
public bool IsEmpty()
{
return firstNode == null;
} // end method IsEmpty

// output List contents
public void Display()
{
if (IsEmpty())
{
Console.WriteLine("Empty " + name);
} // end if
else
{
Console.Write("The " + name + " is: ");
ListNode current = firstNode;
// output current node data while not at end of list
while (current != null)
{
Console.Write(current.Data + " ");
current = current.Next;
} // end while
Console.WriteLine("\n");
} // end else
} // end method Display
} // end class List

// class EmptyListException declaration
public class EmptyListException : Exception
{
// parameterless constructor
public EmptyListException() : base("The list is empty")
{
// empty constructor
} // end EmptyListException constructor

// one-parameter constructor
public EmptyListException(string name)
: base("The " + name + " is empty")
```

```
{
// empty constructor
} // end EmptyListException constructor

// two-parameter constructor
public EmptyListException(string exception, Exception inner)
: base(exception, inner)
{
// empty constructor
} // end EmptyListException constructor
} // end class EmptyListException

// class to test List class functionality
class ListTest
{
public static void Main(string[] args)
{
List list = new List();
// create List container
// create data to store in List
bool aBoolean = true;
char aCharacter = '$';
int anInteger = 34567;
string aString = "hello";
// use List insert methods
list.InsertAtFront(aBoolean);
list.Display();
list.InsertAtFront(aCharacter);
list.Display();
list.InsertAtBack(anInteger);
list.Display();
list.InsertAtBack(aString);
list.Display();
// use List remove methods
object removedObject;
// remove data from list and display after each removal
try
{
removedObject = list.RemoveFromFront();
```

```
Console.WriteLine(removedObject + " removed");
list.Display();
removedObject = list.RemoveFromFront();
Console.WriteLine(removedObject + " removed");
list.Display();
removedObject = list.RemoveFromBack();
Console.WriteLine(removedObject + " removed");
list.Display(); removedObject = list.RemoveFromBack();
Console.WriteLine(removedObject + " removed");
list.Display();
} // end try

catch (EmptyListException emptyListException)
{
Console.Error.WriteLine("\n" + emptyListException);
} // end catch

// Keep the console window open in debug mode.

Console.ReadKey();

} // end Main
} // end class ListTest
} // end namespace LinkedListLibrary
```

سوف نحصل على الخرج التالي:

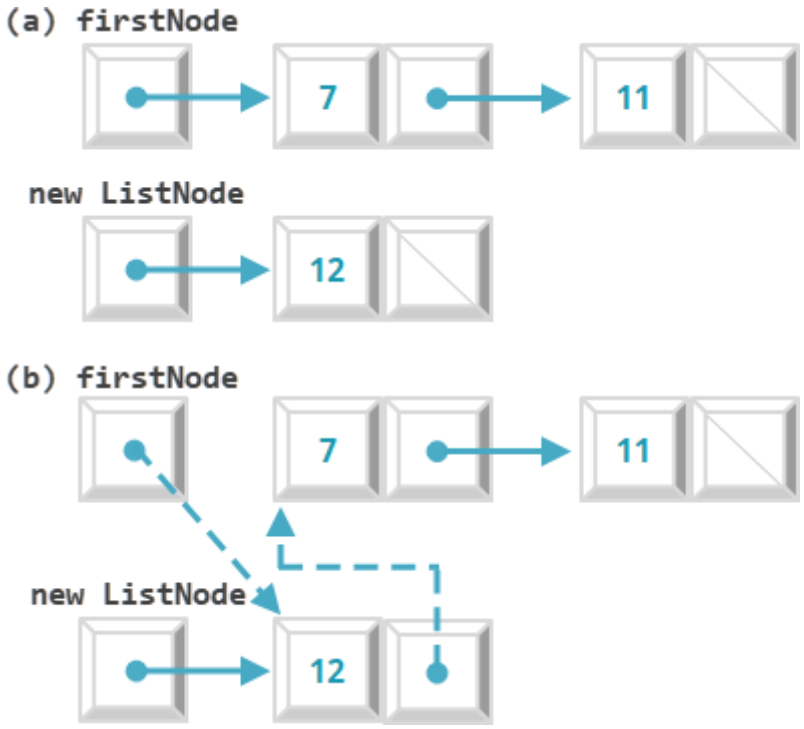
```
The list is: True

The list is: $ True

The list is: $ True 34567

The list is: $ True 34567 hello

 $removed
The list is: True 34567 hello

True removed
The list is: 34567 hello

hello removed
The list is: 34567

 34567removed
Empty list
```

1.2. الطريقة InsertAtFront

تقوم هذه الطريقة بإنشاء غرض وجعل كلّ من مؤشِّر أوّل عنصر ومؤشِّر آخر عنصر يؤشّران عليه إذا كانت القائمة الفارغة. أمّا إذا كانت القائمة غير فارغة فيتمّ إنشاء غرض وجعله يؤشّر على أوّل عنصر من القائمة ومن ثمّ إسناد مؤشّر أوّل عنصر إلى هذا الغرض.

```
if (IsEmpty())
    firstNode = lastNode = new ListNode(insertItem);
else
    firstNode = new ListNode(insertItem, firstNode);
```

ويوضّح الشكل الآتي عملية إضافة عنصر إلى بداية قائمة:

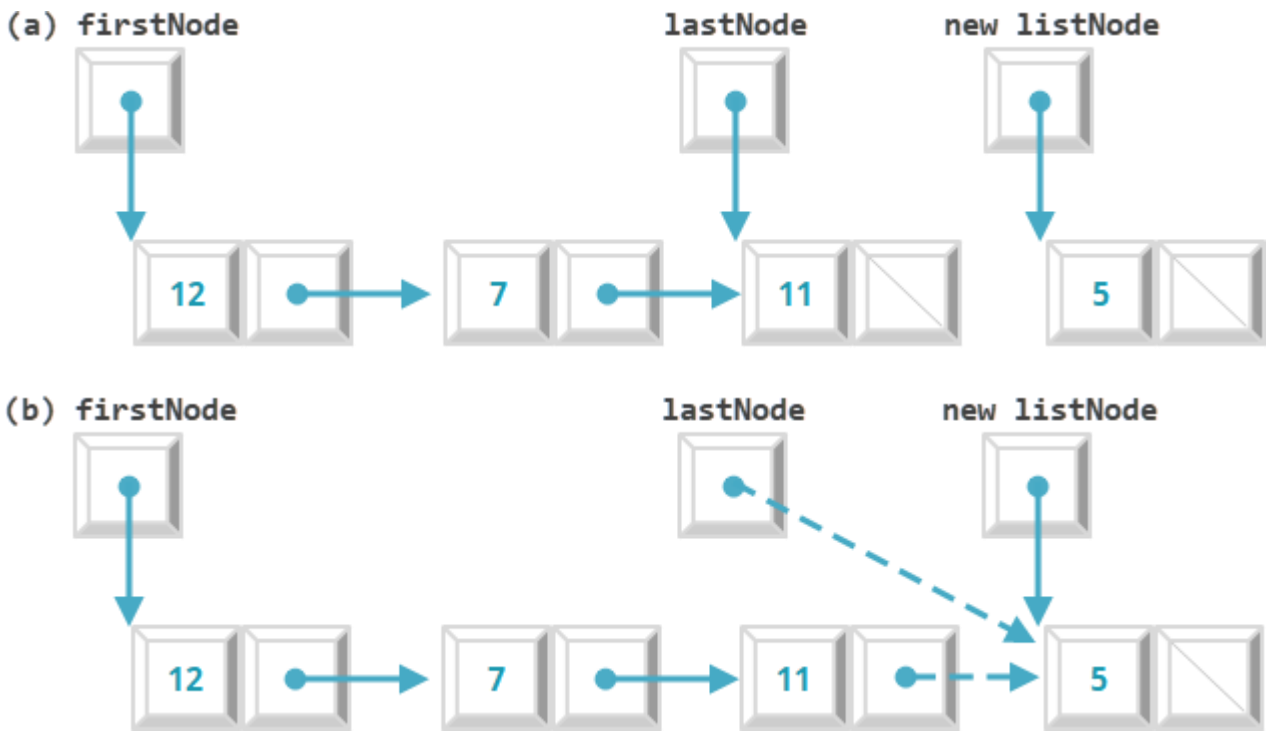


2.2. الطريقة InsertAtBack

تقوم هذه الطريقة بإنشاء غرض وجعل كلّ من مؤشِّر أوّل عنصر ومؤشِّر آخر عنصر يؤشّران عليه إذا كانت القائمة الفارغة. أمّا إذا كانت القائمة غير فارغة فيتمّ إنشاء غرض وجعله المؤشّر التالي لآخر غرض يؤشّر عليه ومن ثمّ جعل مؤشّر آخر عنصر يؤشّر عليه.

```
if (IsEmpty())
    firstNode = lastNode = new ListNode(insertItem);
else
    lastNode = lastNode.Next = new ListNode(insertItem)
```

ويبيّن الشكل التالي كيفية إدراج غرض في آخر قائمة:

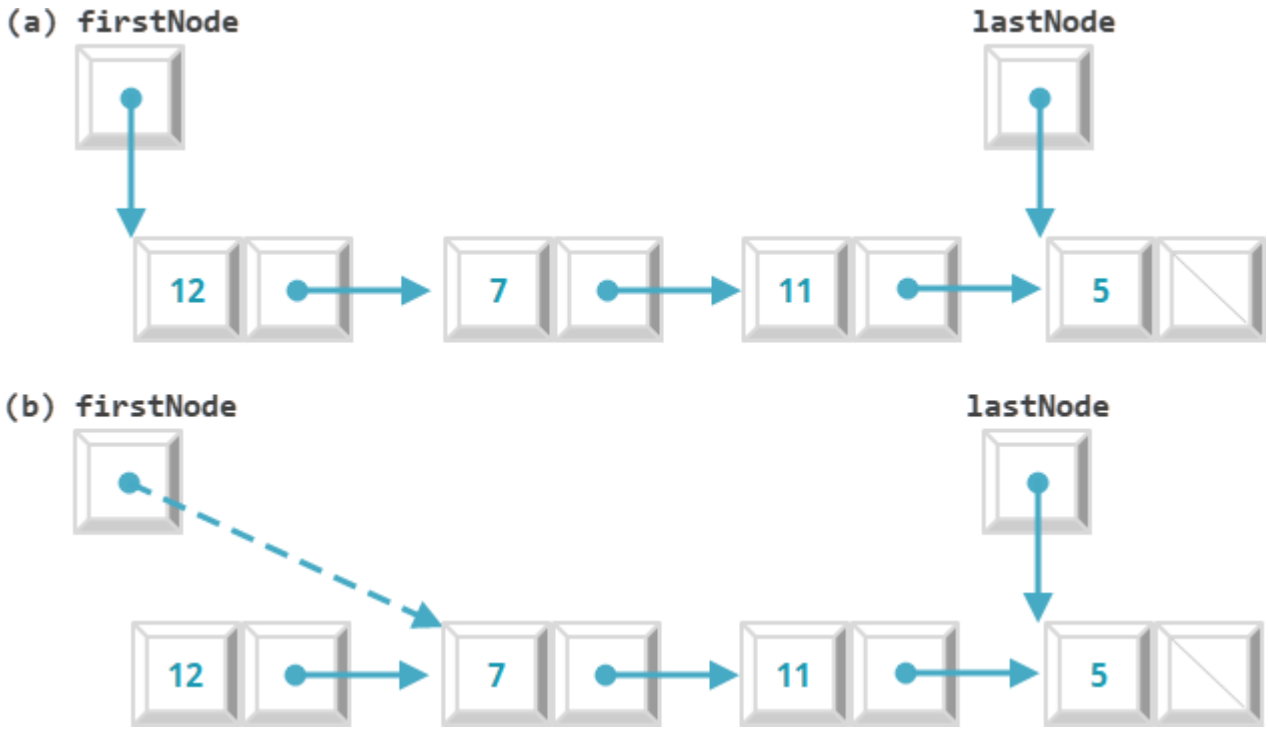


3.2. الطريقة RemoveFromFront

تقوم هذه الطريقة بحفظ قيمة أوّل عنصر لاسترجاعها قبل نهاية عملها ووضع مؤشّر أوّل عنصر على العنصر الثاني. أمّا إذا كانت القائمة فارغة فتقوم الطريقة بقذف استثناء. وفي حال كانت القائمة تحوي عنصراً واحداً فقط، يتمّ إسناد القيمة null لمؤشّري بداية ونهاية القائمة.

```
if (IsEmpty())
    throw new EmptyListException(name);
object removeItem = firstNode.Data; // retrieve data
// reset firstNode and lastNode references
if (firstNode == lastNode) firstNode = lastNode = null;
else firstNode = firstNode.Next;
return removeItem; // return removed data
```

ويوضّح الشكل الآتي عملية حذف عنصر من أوّل القائمة:



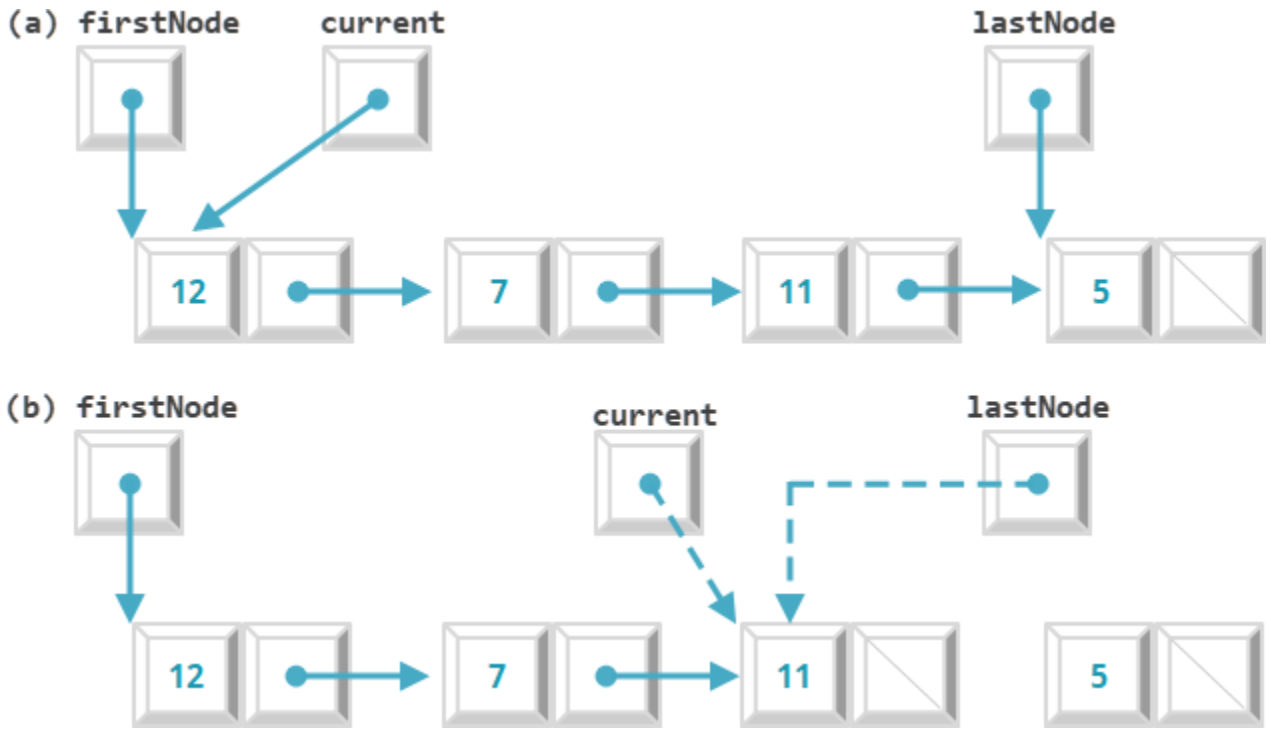
4.2. الطريقة RemoveFromBack

تقوم هذه الطريقة بحفظ قيمة آخر عنصر لاسترجاعها قبل انتهاء عملها. وتقوم بالوصول إلى العنصر ما قبل الأخير من القائمة وإسناد null إلى المؤشّر التالي له ومن ثمّ جعل مؤشر آخر عنصر يؤشّر عليه. أمّا إذا كانت القائمة فارغة، فتقوم هذه الطريقة برفع استثناء. وإذا كانت القائمة تحوي عنصراً واحداً فقط، فتقوم بإسناد القيمة null إلى مؤشّري بداية ونهاية القائمة.

```
if ( IsEmpty() ) throw new EmptyListException(name );
  object removeItem = lastNode.Data; // retrieve data
// reset firstNode and lastNode references  ( firstNode == lastNode )
firstNode = lastNode = null;
else
ListNode current = firstNode;
// loop while current.Next is not lastNode
 while (current.Next != lastNode )
   current = current.Next; // move to next node
  // current is new lastNode
     lastNode = current;      current.Next = null;
```

```
} // end else
return removeItem; // return removed data
```

يبيّن الشكل الآتي كيفية حذف عنصر من نهاية قائمة:



3. الطرائق والصفوف العامة

في كثير من الأحيان، نحتاج إلى تعريف طرائق عامة Generic methods أي أنّها تتعامل مع أي نمط من أنماط المعطيات.

مثال1:

في الرمّاز الآتي، لدينا ثلاث مصفوفات كلّ منها مخصص لتخزين نمط معطيات مختلف عن النمطين الآخرين. ولكتابة عناصر المصفوفات الثلاث، نحتاج إلى كتابة ثلاث طرائق تقوم بنفس العمل:

```
// OverloadedMethods.cs
// Using overloaded methods to display arrays of different types.
 using System;
class OverloadedMethods {
    public static void Main( string[] args ) {

        // create arrays of int, double and char
        int[] intArray = { 1, 2, 3, 4, 5, 6 };
        double[] doubleArray = { 1.1, 2.2, 3.3, 4.4, 5.5, 6.6, 7.7 };
        char[] charArray = { 'H', 'E', 'L', 'L', 'O' };

        Console.WriteLine( "Array intArray contains:" );
        DisplayArray( intArray ); // pass an int array argument

        Console.WriteLine( "Array doubleArray contains:" );
        DisplayArray( doubleArray ); // pass a double array argument

        Console.WriteLine( "Array charArray contains:" );
        DisplayArray( charArray ); // pass a char array argument

        Console.ReadKey();

    } // end Main

    // output int array
private static void DisplayArray( int[] inputArray ) {
        foreach ( int element in inputArray )
            Console.Write( element + " " );
```

```
        Console.WriteLine( "\n" );
    } // end method DisplayArray

    // output double array

private static void DisplayArray( double[] inputArray ) {
        foreach ( double element in inputArray )
            Console.Write( element + " " );
        Console.WriteLine( "\n" );
    } // end method DisplayArray

    // output char array
private static void DisplayArray( char[] inputArray ) {
      foreach ( char element in inputArray )
          Console.Write( element + " " );
      Console.WriteLine( "\n" );
} // end method DisplayArray
} // end class OverloadedMethods
```

وبعد التنفيذ، يظهر الخرج الآتي:

```
Array intArray contains:
1 2 3 4 5 6

Array doubleArray contains:
1.1 2.2 3.3 4.4 5.5 6.6 7.7

Array charArray contains:
H E L L O
```

ويمكن الاستغناء عن الطرائق الثلاث التي تحمل الاسم DisplayArray وتقوم بنفس العمل بالطريقة العامة الموضّحة فيما يأتي:

```
// output array of all types
private static void DisplayArray<T>(T[] inputArray)
{

    foreach (T element in inputArray)
        Console.Write(element + " ");
    Console.WriteLine("\n");

} // end method DisplayArray
```

حيث يشير الحرف (T) على نمط عام ويمكن استبداله بأي حرف آخر. ولا داعٍ لتغيير أي شيء في الطريقة Main حيث يتم استدعاء الطريقة DisplayArray في كل مرة مع مصفوفة من المصفوفات الثلاث كما هو موضح:

```
Console.WriteLine("Array intArray contains:");
DisplayArray(intArray); // pass an int array argument

Console.WriteLine("Array doubleArray contains:");
DisplayArray(doubleArray);
// pass a double array argument

Console.WriteLine("Array charArray contains:");
DisplayArray(charArray);   // pass a char array argument
```

ونلاحظ أنّ عملية الاستدعاء لم تتغيّر مقارنة مع الحالة التي استخدمنا فيها ثلاث مصفوفات. وبعد التنفيذ، سنحصل على نفس الخرج السابق.

مثال2:

يمكن في بعض الحالات، وضع قيود على الأنماط العامة. ففي الرمّاز التالي، نستخدم الطريقة العامة Maximum التي تعيد القيمة العظمى من بين ثلاث قيم تُمرّر إليها كوسائط دخل:

```
// MaximumTest.cs
// Generic method Maximum returns the largest of three objects.
using System;
class MaximumTest
{
    public static void Main()
    {
        Console.WriteLine("Maximum of {0}, {1} and {2} is {3}\n",
              3, 4, 5, Maximum(3, 4, 5));
        Console.WriteLine("Maximum of {0}, {1} and {2} is {3}\n",
                6.6, 8.8, 7.7, Maximum(6.6, 8.8, 7.7));
        Console.WriteLine("Maximum of {0}, {1} and {2} is {3}\n",
   "pear", "apple", "orange", Maximum("pear", "apple", "orange"));
    } // end Main
    // generic function determines the
    // largest of the IComparable objects
    private static T Maximum<T>(T x, T y, T z)
        where T : IComparable<T>
    {
        T max = x; // assume x is initially the largest
        // compare y with max
        if (y.CompareTo(max) > 0) max = y; // y is the largest so far
        // compare z with max
        if (z.CompareTo(max) > 0) max = z; // z is the largest
        return max; // return largest object
    } // end method Maximum
} // end class MaximumTest
```

بما أنّ المعاملان أصغر(<) وأكبر(>) لا يمكن استخدامهما مع جميع الأنماط، فيجب استخدام الطريقة CompareTo المصرّح عنها في الواجهة العامة IComparable<T>. حيث الطريقة x.CompareTo(y) القيمة (0) إذا كان x يساوي y، وتعيد قيمة سالبة إذا كان x أصغر من y، وتعيد قيمة موجبة في حال كان x أكبر من y.

ويتمّ تحديد قيد على النمط T في الطريقة Maximum بأنّه من النمط IComprable كما يلي:

```
where T : IComparable< T >
```

وبعد تنفيذ الرمّاز السابق، نحصل على الخرج:

```
Maximum of 3, 4 and 5 is 5

Maximum of 6.6, 8.8 and 7.7 is 8.8

Maximum of pear, apple and orange is pear
```

مثال3:

ويمكن للصفّ بأكمله أن يكون صفّاً عاماً Generic Class، ويسمح الصفّ العام بتوصيف صفّ على نحو مستقلّ عن أنماط معطيات أعضائه. ففي الرمّاز الآتي، نصرّح عن الصفّ العام GenericClass الذي يحتوي على الحقل العام genericMember والخاصّية العامة GenericProperty والطريقة العامة GenericMethod وبانٍ عام:

```
using System;
 class GenericClass<T> {
    private T genericMember;

    public GenericClass(T value)  {
        genericMember = value;
    }

    public T GenericMethod(T genericParameter) {
        // The "typeof" operator can be applied on a typename to obtain
        // the corresponding object of class Type

        Console.WriteLine("Parameter type: {0}, value: {1}",
                    typeof(T).ToString(), genericParameter);
        Console.WriteLine("Return type: {0}, value: {1}",
                    typeof(T).ToString(), genericMember);

        return genericMember;
    }
```

```
    public T GenericProperty { get; set; }

}// end class GenericClass

class GenericClassTester {
    public static void Main( )  {
        GenericClass<int> intGenericClass = new GenericClass<int>(10);
GenericClass<double> doubleGenericClass = new GenericClass<double>(3.14);
        int val1 = intGenericClass.GenericMethod(200);
        double val2 = doubleGenericClass.GenericMethod(8.55);
        Console.WriteLine(" val1 = " + val1);
        Console.WriteLine(" val2 = " + val2);
        Console.ReadKey();
    }// end Main
}// end class GenericClassTester
```

يقوم الباني بإسناد قيمة وسيط دخل له إلى الحقل العام. وتقوم الطريقة العامة بطباعة نمط معطيات وسيط دخل لها وقيمته، ويتمّ استدعاء نمط وسيط الدخل باستخدام الطريقة typeof(T). وفي الطريقة Main، تمّ إنشاء غرض من الصفّ العام للتعامل مع نمط المعطيات int وتمّت تسميته intGenericClass. وتمّ إنشاء غرض من الصفّ العام للتعامل مع نمط المعطيات double وتمّت تسميته doubleGenericClass. ثمّ تمّ استدعاء الطريقة GenericMethod من قِبَل كلّ من الغرضين من أجل القيم (200) و(8.55) على الترتيب. وأخيراً، تمّت طباعة القيمتين المعادتين من استدعاء الطريقة في المرّتين السابقتين. وبعد التنفيذ، نحصل على الخرج الآتي:

```
Parameter type: System.Int32, value: 200
Return type: System.Int32, value: 10
Parameter type: System.Double, value: 8.55
Return type: System.Double, value: 3.14
 val1 = 10
 val2 = 3.14
```

1.3. الصفّ العام: القائمة المترابطة LinkedList

يمثّل الصفّ العام LinkedList قائمة مترابطة مضاعفة حيث يؤشّر كلّ عنصر على العنصر التالي وعلى العنصر السابق. وتحوي كلّ عقدة من القائمة الخاصّية Value والخاصّيتين (للقراءة فقط) التالي Next والسابق Previous.

من أهم خصائص هذا الصفّ:

First	أوّل عقدة من القائمة
Last	آخر عقدة من القائمة
Previous	العقدة السابقة
Next	العقدة التالية
Value	قيمة العقدة

ومن أهمّ طرائقه:

AddLast	إضافة عنصر إلى آخر القائمة
Find	إعادة العقدة التي تحوي قيمة معينة
Remove	حذف عقدة

مثال:

في الرمّاز الآتي، نبيّن كيفية استخدامه الصفّ العام LinkedList:

```
// LinkedListTest.cs
// Using LinkedLists.
using System;
using System.Collections.Generic;
public class LinkedListTest
{
    private static readonly string[] colors = { "black", "yellow",
                    "green", "blue", "violet", "silver" };
    private static readonly string[] colors2 = { "gold", "white",
                     "brown", "blue", "gray" };
    // set up and manipulate LinkedList objects
    public static void Main()
    {
```

```
        LinkedList<string> list1 = new LinkedList<string>();
        // add elements to first linked list
        foreach (var color in colors)
            list1.AddLast(color);
        // add elements to second linked list via constructor
        LinkedList<string> list2 = new LinkedList<string>(colors2);
        Concatenate(list1, list2); // concatenate list2 onto list1
        PrintList(list1); // display list1 elements
        Console.WriteLine("\nConverting strings in list1 to uppercase\n");
        ToUppercaseStrings(list1); // convert to uppercase string
        PrintList(list1); // display list1 elements
        Console.WriteLine("\nDeleting strings between BLACK and BROWN\n");
        RemoveItemsBetween(list1, "BLACK", "BROWN");
        PrintList(list1);  // display list1 elements
        PrintReversedList(list1);
        // display list in reverse order
    } // end Main

    // display list contents
    private static void PrintList<T>(LinkedList<T> list)
    {
        Console.WriteLine("Linked list: ");
        foreach (T value in list)
            Console.Write("{0} ", value);
        Console.WriteLine();
    } // end method PrintList
    // concatenate the second list on the end of the first list
    private static void Concatenate<T>(LinkedList<T> list1,
                        LinkedList<T> list2)
    {
        // concatenate lists by copying element values
        // in order from the second list to the first list
        foreach (T value in list2) list1.AddLast(value);
        // add new node
    } // end method Concatenate

    // locate string objects and convert to uppercase
    private static void ToUppercaseStrings(LinkedList<string> list)
    {
```

```
        // iterate over the list by using the nodes
        LinkedListNode<string> currentNode = list.First;
        while (currentNode != null)
        {
            string color = currentNode.Value; // get value in node
            currentNode.Value = color.ToUpper(); // convert to uppercase
            currentNode = currentNode.Next; // get next node
        } // end while
    } // end method ToUppercaseStrings
    // delete list items between two given items
    private static void RemoveItemsBetween<T>(LinkedList<T> list,
                                T startItem, T endItem)
    {
        // get the nodes corresponding to the start and end item
        LinkedListNode<T> currentNode = list.Find(startItem);
        LinkedListNode<T> endNode = list.Find(endItem);
        // remove items after the start item
        // until we find the last item or the end of the linked list
        while ((currentNode.Next != null) &&
                  (currentNode.Next != endNode))
        {
            list.Remove(currentNode.Next); // remove next node
        } // end while
    } // end method RemoveItemsBetween

    // display reversed list
    private static void PrintReversedList<T>(LinkedList<T> list)
    {
        Console.WriteLine("Reversed List:");
        // iterate over the list by using the nodes
        LinkedListNode<T> currentNode = list.Last;
        while (currentNode != null)
        {
            Console.Write("{0} ", currentNode.Value);
            currentNode = currentNode.Previous; // get previous node
        } // end while Console.WriteLine();
    } // end method PrintReversedList
} // end class LinkedListTest
```

ويتّضح من خلال الرمّاز سهولة إضافة وحذف عنصر من قائمة مترابطة بالمقارنة مع الحالة السابقة التي لم نستخدم فيها الصفّ العام LinkedList. وبعد التنفيذ، نحصل على الخرج الآتي:

```
Linked list:
black yellow green blue violet silver gold white brown blue gray

Converting strings in list1 to uppercase

Linked list:
BLACK YELLOW GREEN BLUE VIOLET SILVER GOLD WHITE BROWN BLUE GRAY

Deleting strings between BLACK and BROWN

Linked list:
BLACK BROWN BLUE GRAY
Reversed List:
GRAY BLUE BROWN BLACK  Press any key to continue ...
```

خاتمة

توجد العديد من بنى المعطيات ذات الاستخدامات الخاصة في لغة C#، وسنستعرض بعضها في الفصل القادم.

الأنشطة المرافقة

التمرين الأوّل:

دمج قائمتين مترابطتين Merging two Lists

اكتب برنامجاً يقوم بدمج قائمتين من الأعداد الطبيعية المرتّبة تصاعدياً ضمن قائمة واحدة مرتّبة.

التمرين الثاني:

المصفوفة العامة GenericArray

عرّف الصفّ GenericArray الذي يحوي مصفوفة من الأغراض العامة ويحوي على:

- الطريقة العامة Add التي تسمح بإضافة عنصر إلى المصفوفة.
- الطريقة العامة Swap التي تسمح بالمبادلة بين قيمتي عنصرين من عناصر المصفوفة.
- الطريقة العامة Search التي تسمح بالبحث التسلسلي في المصفوفة لإرجاع دليل عنصر موجود في المصفوفة، وفي حال عدم العثور عليه تعيد القيمة(−1).

ثمّ عرِّف الصفّ Gtest الذي يسمح باختبار الصفّ GenericArray.

التمرين الثالث:

صفّ المكدّس Stack

المكدّس Stack هو بنية معطيات تسمح بتخزين عناصر من أي نمط، وهو أشبه بعلبة يتمّ الوصول إلى العناصر المخزّنة فيها بشكل تسلسلي. ولاسترجاع عنصر من وسط المكدّس، لا بُدّ من إخراج جميع العناصر التي تقع فوقه ويتمّ الإخراج وفق مبدأ "الداخل أخيراً هو الخارج أوّلاً".

1. قم بإنشاء الصفّ العام Stack والذي يضمّ العناصر الآتية:
   - الحقل top للتأشير على العنصر الموجود في قمة المكدّس
   - مصفوفة العناصر وهي عامة
   - بانٍ يأخذ كوسيط دخل السعة العظمى للمكدّس
   - بانٍ افتراضي لإنشاء مكدّس بعشرة عناصر
   - الطريقة Push لإدخال عنصر إلى المكدّس، وتقذف استثناء إذا كان المكدّس مليئاً بالعناصر
   - الطريقة Pop لإخراج عنصر من المكدّس، وتقذف استثناء إذا كان المكدّس فارغاً
2. قم بإنشاء الصفّ Tester الحاوي على الطريقة Main وقم بإنشاء مكدّس لعناصر من النمط int ومكدّس لعناصر من النمط double واختبر الطرائق التي قمت بتعريفها.

التمرين الرابع:

صفّ الأسماء LinkedListNoDuplicates

قم بإنشاء الصفّ LinkedListNoDuplicates الذي يسمح يقوم بقراءة مجموعة أسماء ويقوم بتخزينها ضمن قائمة مترابطة بحيث لا تحتوي القائمة على أي أسماء مكرّرة. ثم عرِّف الصفّ Tester الذي يسمح باختبار الصفّ السابق.

التمرين الخامس:

ReversingLinkedList

قم بإنشاء الصفّ ReversingLinkedList الذي يقوم بقراءة مجموعة محارف ويقوم بتخزينها ضمن قائمة مترابطة ثمّ يقوم بنسخ عناصرها إلى قائمة أخرى بترتيب معكوس. ثم عرِّف الصفّ Tester الذي يسمح باختبار الصفّ السابق.

المراجع

1. "التصميم والبرمجة غرضية التوجّه"، الدكتور سامي خيمي، الإجازة في تقانة المعلومات، من منشورات الجامعة الافتراضية السورية، الجمهورية العربية السورية، 2018.