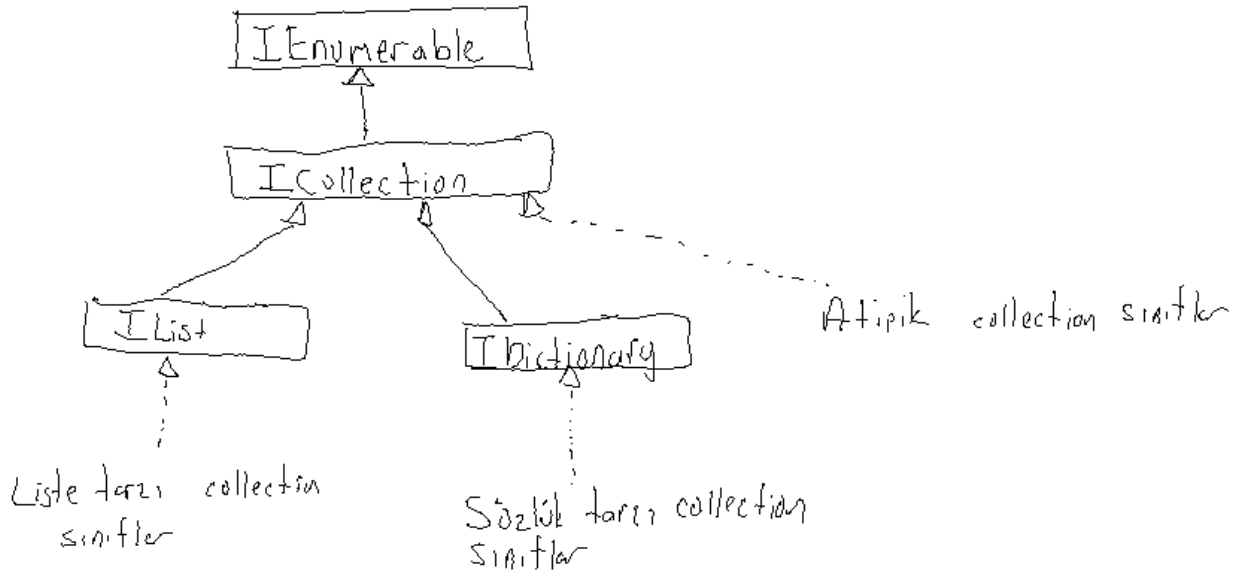
1. .NET’te Collection Sınıflar

.NET'te collection sınıflar üç gruba ayrılmaktadır:

1) Liste tarzı collection sınıflar
2) Sözlük tarzı collection sınıflar
3) Atipik collection sınıflar

.NET'teki collection sınıflar bazı arayüzleri desteklemektedir. Bu nedenle önce o arayüzlerin ele alınması anlatımı kolaylaştıracaktır.



IEnumerable arayüzünün GetEnumerator isimli tek bir metodu vardır:

```
interface IEnumerable
{
      IEnumerator GetEnumerator();
}
```

GetEnumerator metodunun geri dönüş değeri **IEnumerator** türünden bir arayüz türündendir. **IEnumerator** arayüzü bir collection'ı türden bağımsız olarak dolaşmak için kullanılmaktadır. Bu arayüzün üç elemanı vardır: **MoveNext**, **Current** ve **Reset**.

Enumerator bir imleç mekanizmasıyla collection’ın elemanları üzerinde gezinmemizi sağlar. **MoveNext** metodu imleci sonraki elemana kaydırmaktadır. **Current** property'si object türündendir ve read-only'dir. Enumerator ilk alındığında imleç ilk elemanın bir gerisindedir. Yani bizim imleci

ilk elemana konumlandırmamız için bir kez **MoveNext** yapmamaız gerekir. **MoveNext** metodunun parametrik yapısı şöyledir:

```
bool MoveNext()
```

MoveNext her çağrıldığında imleç bir sonraki elemana konumlanır. MoveNext ile imleç eğer son elemandan sonraya konumlandırılmaya çalışılırsa MoveNext false ile geri döner. Bu da collection'ın sonuna geldiğimizi gösterir. Bu durumda collection'ın sonuna kadar dolaşmak için şöyle basit bir while döngüsü yeterlidir.

```
IEmumerator ie = col.GetEnumerator();

while (ie.MoveNext())
{
      //...
}
```

Reset metodu imleci ilk durumuna (yani ilk elemanın bir gerisine) konumlandırmaktadır. **Current** property'si imleç o anda hangi elemana konumlandırılmışsa onu bize object olarak verir.

IEnumerable arayüzü sayesinde biz her türlü collection sınıfı onun özelliklerini bilmeden, türden bağımsız olarak dolaşabiliriz. Genel dolaşım kalıbı aşağıdaki gibi olabilir:

```
using System;
using System.Collections;

namespace BM
{
    class App
    {
        public static void Main()
        {
            ArrayList al = new ArrayList();

            for (int i = 0; i < 10; ++i)
                al.Add(i * 10);

            IEnumerator ie = al.GetEnumerator();
            while (ie.MoveNext())
            {
                int val = (int)ie.Current;
                Console.Write("{0} ", val);
            }
            Console.WriteLine();

            ie.Reset();
```

```
            while (ie.MoveNext())
            {
                int val = (int)ie.Current;
                Console.Write("{0} ", val);
            }
            Console.WriteLine();
        }
    }
}
```

Ekran Çıktısı:

```
0 10 20 30 40 50 60 70 80 90
0 10 20 30 40 50 60 70 80 90
```

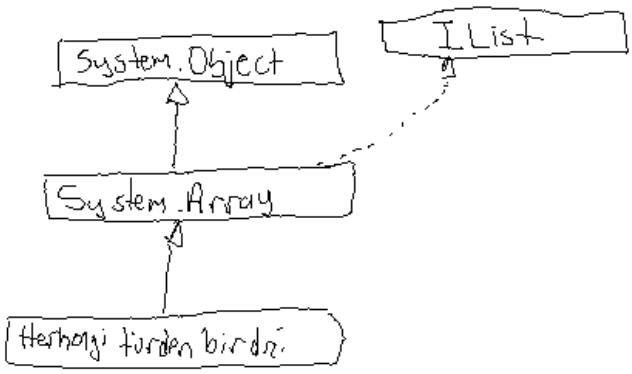
Aşağıdaki örnekte Walk isimli metot IEnumerable arayüzü türünden bir referansı parametre olarak almaktadır. Biz de bu metoda o arayüzü destekleyen herhangi bir sınıf referansını ya da yapı nesnesini parametre olarak geçirebiliriz:

```
using System;
using System.Collections;

namespace BM
{
    class App
    {
        public static void Main()
        {
            ArrayList al = new ArrayList();

            for (int i = 0; i < 10; ++i)
                al.Add(i * 10);

            Sample.Walk(al);
        }
    }
    class Sample
    {
        public static void Walk(IEnumerable ie)
        {
            IEnumerator ien = ie.GetEnumerator();

            while (ien.MoveNext())
                Console.Write("{0} ", (int)ien.Current);
            Console.WriteLine();
        }
    }
}
```

Ekran Çıktısı:

```
0 10 20 30 40 50 60 70 80 90
```

C#'ta tüm diziler de sanki liste tarzı bir collection sınıf gibidir. Şöyle ki: C# standartlarına göre tüm dizilerin **System.Array** isimli bir sınıftan türetildiği varsayılmaktadır. System.Array sınıfı da IList arayüzünü destekler ve System.Object sınıfından türetilmiştir:



Örneğin:

```
using System;
using System.Collections;

namespace BM
{
    class App
    {
        public static void Main()
        {
            double[] d = new double[] { 1, 2, 3, 4.5 };
            int[] i = new int[] { 10, 20, 30, 40, 50 };
            string[] s = new string[] { "Hülya", "Ahmet", "Harun", "Murat", "Fatma"
};
            ArrayList al = new ArrayList();

            al.Add("Bursa");
            al.Add("İzmir");
            al.Add("Adana");

            Sample.Walk(d);
            Sample.Walk(i);
            Sample.Walk(s);
            Sample.Walk(al);

        }
    }
    class Sample
    {
```

```
        public static void Walk(IEnumerable ie)
        {
            IEnumerator ien = ie.GetEnumerator();

            while (ien.MoveNext())
                Console.Write("{0} ", ien.Current.ToString());
            Console.WriteLine();
        }
    }

}
```

Ekran Çıktısı:

1 2 3 4,5
10 20 30 40 50
Hülya Ahmet Harun Murat Fatma
Bursa İzmir Adana

C#'taki **foreach** deyimi aslında kendi içerisinde IEnumearable arayüzüyle dolaşımı yapmaktadır. Yani biz foreach deyimini kullandığımızda, derleyicinin ürettiği koda bakarsak onun dolaşımı IEnumerable arayüzü ile yaptığını görürüz. Başka bir deyişle T bir tür olmak üzere:

```
foreach (T x in a)
{
      //...
}
```

Döngüsü aşağıdakiyle eşdeğerdir:

```
IEnumerator ien = a.GetEnumerator();
while (ien.MoveNext())
{
      T x = (T)ien.Current;
      //...
}
```

Pekiyi biz IEnumerable arayüzünü destekleyen bir sınıf yazabilir miyiz? Bunun için şöyle bir örnek verilebilir:

```
using System;
using System.Collections;
```

```
namespace BM
{
    class App
    {
        public static void Main()
        {
            MyCollection mc = new MyCollection(10, 20);
            IEnumerator ien = mc.GetEnumerator();

            while (ien.MoveNext())
            {
                int val = (int)ien.Current;
                Console.Write("{0} ", val);
            }
            Console.WriteLine();

            foreach (int x in mc)
                Console.Write("{0} ", x);
            Console.WriteLine();
        }
    }

    class MyCollection: IEnumerable
    {
        private int m_low;
        private int m_high;
        public MyCollection(int low, int high)
        {
            m_low = low;
            m_high = high;
        }

        public IEnumerator GetEnumerator()
        {
            return new MyEnumerator(this);
        }

        private class MyEnumerator: IEnumerator
        {
            private MyCollection m_mc;
            private int m_curVal;

            public MyEnumerator(MyCollection mc)
            {
                m_mc = mc;
                m_curVal = mc.m_low - 1;
            }

            public bool MoveNext()
            {
                if (m_curVal == m_mc.m_high)
```

```
                    return false;

                ++m_curVal;

                return true;
            }

            public object Current
            {
                get { return m_curVal; }
            }

            public void Reset()
            {
                m_curVal = m_mc.m_low - 1;
            }
        }
    }
}
```

Ekran Çıktısı:
10 11 12 13 14 15 16 17 18 19 20
10 11 12 13 14 15 16 17 18 19 20

Anahtar Notlar: Alçak seviyeli bir dilden yüksek seviyeli bir dile dönüştürme yapan tersine mühendislik programlarına "**decompiler**" denilmektedir. Örneğin .NET'in .exe ve .dll dosyalarını C#'a dönüştüren programlar birer "decompiler" programlardır. .NET için ilk decompiler'lardan olan "Salamander" paralı üründü. Sonra "Reflector" isimli "open source" bir program kullanılmaya başlandı. Ancak "reflector"ü yazanlar "Redgate" bunu firmasına sattılar. Tabii eski "open source" kodlara erişim devam etti. Yeni bir proje grubu da "ILSpy" ismiyle bunu sürdürdü. Maalesef .NET gibi Java gibi dillerin arakodları "decompile" edilebilmektedir. Ancak C/C++ gibi dillerle saf makina diline dönüştürülmüş kodlar halen etkin bir biçimde "decompile" edilememektedir.

1.1. ICollection Arayüzü

ICollection arayüzünün dört elemanı vardır: **CopyTo**, **Count**, **IsSynchronized**, **SyncRoot**. CopyTo metodu collection sınıf içerisindekileri bir diziye aktarmak için kullanılır. Metodun parametrik yapısı şöyledir:

```
void CopyTo(Array array,int index)
```

Metodun birinci parametresi aktarımın yapılacağı diziyi, ikinci parametresi bu dizide kopyalamanın hangi indeksten itibaren yapılacağıdır. Arayüzün Count isimli int türden read only property'si collection'daki eleman sayısını bize verir.

Örneğin:

```
using System;
using System.Collections;
```

```
namespace BM
{
    class App
    {
        public static void Main()
        {
            ArrayList al = new ArrayList();

            for (int i = 0; i < 10; ++i)
                al.Add(i);
            Foo(al);

        }
        public static void Foo(ICollection ic)
        {
            int[] a = new int[ic.Count];
            ic.CopyTo(a, 0);

            foreach (int x in a)
                Console.Write("{0} ", x);
            Console.WriteLine();
        }

    }
}
```

Ekran Çıktısı:
0 1 2 3 4 5 6 7 8 9

Burada aktarılacak dizinin oradaki elemanın türüyle aynı türden olması gerekir. Aksi halde exception oluşur.

ICollection arayüzünün Count isimli read-only property elemanı collection içerisindeki eleman sayısını vermektedir. IsSynchronized property'si collection'ın thread güvenli (thread safe) olup olmadığı bilgisini bize verir. SyncRoot property'si yine thread senkronizasyonu için kullanılan bir property'dir.

IList Arayüzü

IList arayüzü liste tarzı collection sınıfların desteklediği ortak bir arayüzdür. İçerisinde pek çok önemli eleman vardır. Bu elemanların hepsi bu arayüzü destekleyen sınıflarda bulunmak zorundadır.

Add metodu collection'ın sonuna eleman eklemek için kullanılır:

```
int Add(Object value)
```

Metot eklenecek elemanı bizden parametre olarak alır. Eklemenin yapıldığı indekse geri döner.

Anahtar Notlar: Bilindiği gibi bir dizeye ekleme yapılamaz. Pekiyi hani diziler Array sınıfından türetilmişti de o sınıf da IList arayüzünü destekliyordu? Bu durumda biz bir dizi için Add yapabilir miyiz? İşte Array sınıfı içerisinde Add ve birkaç metot açıkça (explicit) desteklenemiştir. Array sınıfının Add metodunda doğrudan exception (NotSupportedException) throw edilmektedir. Yani biz istesek de diziye bu metotlarla ekleme yaapamayız.

IList arayüzünün Clear metodu collection'daki tüm elemanları silmektedir. Contains metodu belli bir eleman colleciton'da var mı diye bakar. Örneğin:

```
using System;
using System.Collections;

namespace BM
{
    class App
    {
        public static void Main()
        {
            ArrayList al = new ArrayList();
            al.Add("Gülsüm");
            al.Add("Ali");
            al.Add("Zeynep");

            Console.WriteLine(al.Contains("Ali") ? "Var" : "Yok");
            Console.WriteLine(al.Contains("Sacit") ? "Var" : "Yok");

        }

    }
}
```

Ekran Çıktısı:

Var
Yok

IndexOf da benzer bir arama yapar fakat elemanı bulursa onun collection'daki indeks numarasına bulamasa -1 değerine geri döner. Örneğin:

```
using System;
using System.Collections;

namespace BM
{
    class App
    {
        public static void Main()
        {
            ArrayList al = new ArrayList();
            al.Add("Gülsüm");
            al.Add("Ali");
            al.Add("Zeynep");
```

```
            int index;

            if ((index = al.IndexOf("Ali")) == -1)
                Console.WriteLine("Eleman yok!");
            else
                Console.WriteLine("Eleman {0} indeksinde bulundu", index);

        }

    }
}
```

Ekran Çıktısı:
Eleman 1 indeksinde bulundu

Insert metodu diğer elemanları kaydırarak belli bir indekse ekleme yapmaktadır. Metodun parametrik yapısı şöyledir:

```
void Insert(int index, Object value)
```

Örneğin:

```
using System;
using System.Collections;

namespace BM
{
    class App
    {
        public static void Main()
        {
            ArrayList al = new ArrayList();
            for (int i = 0; i < 10; ++i)
                al.Add(i);
            al.Insert(5, 100);

            foreach (int x in al)
                Console.Write("{0} ", x);
            Console.WriteLine();

        }

    }
}
```

Ekran Çıktısı:
0 1 2 3 4 100 5 6 7 8 9

Collection'ın **Remove** metodu elemanı arar, bulursa onu siler. Eleman collection içerisinde birden fazla yerde varsa Remove yalnızca ilk bulduğu yerdeki elemanı silmektedir. **RemoveAt** ise belli

bir indeksteki elemanı silmekte kullanılır. Silme sırasında collection'daki elemanlar birer kaydırılır (shrink işlemi).
Örneğin:

```
using System;
using System.Collections;

namespace BM
{
    class App
    {
        public static void Main()
        {
            ArrayList al = new ArrayList();
            for (int i = 0; i < 10; ++i)
                al.Add(i * 10);

            al.Remove(30);

            foreach (int x in al)
                Console.Write("{0} ", x);
            Console.WriteLine();

            al.RemoveAt(3);

            foreach (int x in al)
                Console.Write("{0} ", x);
            Console.WriteLine();

        }

    }
}
```

```
Ekran Çıktısı:
0 10 20 40 50 60 70 80 90
0 10 20 50 60 70 80 90
```

IList arayüzünün object türünden read/write indeskleyicisi vardır. Yani her IList arayüzünü destekleyen collection sınıf türünden referansı ya da yapı değişkenini biz köşeli parantez operatörüyle kullanabiliriz. IList arayüzünün **IsFixedSize** property'si collection'ın sabit uzunlukta olup olmadığını belirlemek için, **IsReadOnly** property elemanı da collection'ın read-only olup olmadığını belirlemek için kullanılmaktadır.

IList arayüzünü destekleyen en önemli sınıf ArrayList ve bunun generic biçimi olan List<T> sınıfıdır. İleride GUI işlemlerinde bu arayüzü destekleyen pek çok collection sınıfla karşılaşacağız.

IDictionary Arayüzü

Sözlük tarzı collection sınıflar **IDictionary** arayüzünü desteklemektedir. Bu collection sınıflar anahtar-değer (key-value) çiftlerini bizden alarak kendi içlerinde saklarlar. Sonra onlara anahtarı

verdiğimizde bize o anahtara karşı gelen değeri verirler. Tabii bu işlemi özel algoritmik yöntemlerle çok hızlı yapılmaktadır.

IDictionary arayüzünün Add metodu collection'a eleman eklemek için kullanılır:

```
void Add(Object key, Object value)
```

Metodun birinci parametresi anahtarı (key) ikinci parametresi değeri (value) belirtir. **Clear** metodu collection'daki tüm elemanları silmektedir. **Contains** metodu algoritmik olarak belli bir anahtarın collection'da olup olmadığına bakar. **Remove** ise belli bir anahtarı parametre olarak alır ve onu colelction'dan siler.

IDictionary arayüzünün object parametreli object türünden indeksleyicisi belli bir anahtara karşılık bize değeri vermektedir. Bu indeksleyici read/write biçimdedir. Yani bu indeksleyiciye atama yapıldığında eğer söz konusu anahtar collection'da yoksa indeksleyici bunu aynı zamanda eklemektedir. Arayüzün Keys isimli property elemanı bize tüm anahtarları, Values property'si tüm değerleri ICollection arayüzü ile verir.

Genel olarak IDictionary arayüzünü destekleyen sınıflarda Add metodu ile aynı anahtarı ikinci kez ekleyemeyiz. Bu durumda Exception oluşur. Ancak indeskleyici yoluyla ekleme yapmak istersek eski değer gider onun yerine yeni değer atanır.

Hashtable Sınıfı

En çok kullanılan sözlük tarzı collection sınıf Hashtable sınıfıdır. Bu sınıf "hash tablosu" ya da "hashing" denilen algoritmik yöntemi kullanmaktadır. Örneğin:

```
using System;
using System.Collections;

namespace BM
{
    class App
    {
        public static void Main()
        {
            Hashtable ht = new Hashtable();

            ht.Add("Koray Aki", 123);
            ht.Add("Hülya Aydın", 512);
            ht.Add("Mert Ergin", 512);

            int val = (int)ht["Koray Aki"];
            Console.WriteLine(val);

            ht["Hakan Düzgün"] = 654;
            ht["Hakan Düzgün"] = 657;
```

```
            val = (int)ht["Hakan Düzgün"];
            Console.WriteLine(val);

            foreach (string key in ht.Keys)
                Console.Write("{0} ", key);
            Console.WriteLine();

            foreach (int v in ht.Values)
                Console.Write("{0} ", v);
            Console.WriteLine();
        }

    }
}
```

Ekran Çıktısı:
123
657
Hakan Düzgün Koray Aki Hülya Aydın Mert Ergin
657 123 512 512

Hashtable eleman sayısı az olmayan durumlarda (en az 20, fakat daha fazla olsa daha iyi) ve arama işleminin çok fazla yapıldığı sistemlerde tercih edilmektedir.

Eğer eleman sayısı çok azsa (tipik olarak 20'den az) bu durumda **SortedList** isimli collection tercih edilmelidir. SortedList iki paralel dizi biçiminde gerçekleştirilmiştir. Dolayısıyla anahtar dizisinde eleman sıralı aramayla bulunur. Sonra da değer dizisinde bunun karşılığı olan değer verilir. Eleman sayısı az olduğu için sıralı arama çok etkindir.

Örneğin:

```
using System;
using System.Collections;

namespace BM
{
    class App
    {
        public static void Main()
        {
            SortedList sl = new SortedList();

            sl.Add("Koray Aki", 123);
            sl.Add("Hülya Aydın", 512);
            sl.Add("Mert Ergin", 512);

            int val = (int)sl["Koray Aki"];
            Console.WriteLine(val);

            sl["Hakan Düzgün"] = 654;
            sl["Hakan Düzgün"] = 657;
```

```
            val = (int)sl["Hakan Düzgün"];
            Console.WriteLine(val);

            foreach (string key in sl.Keys)
                Console.Write("{0} ", key);
            Console.WriteLine();

            foreach (int v in sl.Values)
                Console.Write("{0} ", v);
            Console.WriteLine();
        }

    }
}
```

Ekran Çıktısı:

123
657
Hakan Düzgün Hülya Aydın Koray Aki Mert Ergin
657 512 123 512

Hashtable sınıfının generic versiyonun ismi **Dictionary<Key, Value>** sınıfıdır. Ayrıca bir de **HashSet<T>** biçiminde bir versiyonu da vardır. HashSet<T> sınıfı bizden ekleme sırasında anahtar ve değeri ayrı ayrı istemez. Tek bir nesne olarak ister. Onu bulurken eşitlik karşılaştırması yapmaktadır. Dictionary<Key, Value> ise Hashtable gibi kullanılır. Örneğin:

```
using System;
using System.Collections.Generic;

namespace BM
{
    class App
    {
        public static void Main()
        {
            Dictionary<string, int> dict = new Dictionary<string, int>();

            dict.Add("Koray Aki", 123);
            dict.Add("Kaan Kara", 512);
            dict.Add("Hande Karlı", 512);

            int val = dict["Koray Aki"];
            Console.WriteLine(val);

            dict["Murat Koca"] = 654;
            dict["Murat Koca"] = 657;

            val = dict["Murat Koca"];
            Console.WriteLine(val);

            foreach (string key in dict.Keys)
                Console.Write("{0} ", key);
            Console.WriteLine();
```

```
            foreach (int v in dict.Values)
                Console.Write("{0} ", v);
            Console.WriteLine();

        }

    }
}
```

Ekran Çıktısı:

123
657
Koray Aki Kaan Kara Hande Karlı Murat Koca
123 512 512 657

Burada artık Add metodunun parametrelerinin string ve int türünden olduğuna ve indeksleyicinin de artık string türünden olduğuna dikkat ediniz.

SortedList sınıfının da generic biçimi yine SortedList ismiyle bulunmaktadır. Örneğin:

```
using System;
using System.Collections.Generic;

namespace BM
{
    class App
    {
        public static void Main()
        {
            SortedList<string, int> sl = new SortedList<string, int>();

            sl.Add("Koray Aki", 123);
            sl.Add("Kaan Kara", 512);
            sl.Add("Hande Karlı", 512);

            int val = sl["Koray Aki"];
            Console.WriteLine(val);

            sl["Murat Koca"] = 654;
            sl["Murat Koca"] = 657;

            val = sl["Murat Koca"];
            Console.WriteLine(val);

            foreach (string key in sl.Keys)
                Console.Write("{0} ", key);
            Console.WriteLine();

            foreach (int v in sl.Values)
                Console.Write("{0} ", v);
            Console.WriteLine();

        }
```

```
    }
}
```

Ekran Çıktısı:
123
657
Hande Karlı Kaan Kara Koray Aki Murat Koca
512 512 123 657

SortedList<Key, Value> sınıfı System.dll içerisinde bulunmaktadır. Bu nedenle bu sınıf kullanılmadan önce bu DLL'e referans edilmesi gerekir.

Queue Sınıfı

Queue sınıfı kuyruk denilen veri yapısını temsil eden atipik bir collection sınıftır. Doğrudan ICollection arayüzünü desteklemektedir. Kuyruklar FIFO prensibiyle çalışan veri yapılarıdır. Kuyruk için iki önemli işlem söz konusudur: Kuyruğa eleman yerleştirmek ve kuyruktan eleman almak. Örneğin biz X, Y ve Z elemanlarını kuyruğa yerleştirmiş olalım. Bunları almak istediğimizde yine X, Y ve Z sırasına göre alırız. Kuyruk veri yapısı bilgilerin geçici süre sırası bozulmadan bekletilmesi gerektiği sistemlerde kullanılmaktadır. Queue sınıfının Queue<T> biçiminde generic bir versiyonu da vardır.

Kuruğa eleman yerleştirmek için **Enqueue** metodu, eleman almak için **Dequeue** metodu kullanılır:

```
public virtual void Enqueue(Object o)
public virtual Object Dequeue()
```

Count property'si kuyrukta o anda bulunan eleman sayısını bize verir. Kuyruk boşken Dequeue yapılmak istenirse exception oluşur. Örneğin:

```
using System;
using System.Collections;

namespace BM
{
    class App
    {
       public static void Main()
       {
              Queue q = new Queue();

              for (int i = 0; i < 10; ++i)
             q.Enqueue(i);

              while (q.Count > 0)
              {
              int val = (int)q.Dequeue();
```

```
            Console.Write("{0} ", val);
            }
            Console.WriteLine();
        }
    }
}
```

Ekran Çıktısı:
0 1 2 3 4 5 6 7 8 9

Queue sınıfının Peek metodu kuyruğun önündeki elemanı bize verir fakat onu kuyruktan atmaz. Aynı örneği Qeueu sınıfının geeneric biçimi olan Queue<T> ile de yapabiliriz:

```
using System;
using System.Collections.Generic;

namespace BM
{
class App
    {
    public static void Main()
    {
            Queue<int> q = new Queue<int>();

            for (int i = 0; i < 10; ++i)
                q.Enqueue(i);

            while (q.Count > 0)
            {
                int val = q.Dequeue();
                Console.Write("{0} ", val);
            }
            Console.WriteLine();

        }
    }
}
```

Ekran Çıktısı:
0 1 2 3 4 5 6 7 8 9

Artık burada Enqueue metodunun parametresi ve Dequeue metodunun geri dönüş değeri object türünden değil int türdendir.

Diğer bir kuyruk sistemine de öncelik kuyruğu (priority queue) denilmektedir. Bu tür kuyruklara elemanlar bir öncelik derecesi verilerek eklenir. Eleman alınırken en yüksek önceliğe sahip eleman alınır. Maalesef .NET kütüphensinde Microsoft tarafından gerçekleştirilmiş böyle bir sınıf hazır biçimde yoktur.

Stack Sınıfı

LIFO (Last In First Out) prensibiyle çalışan kuyruk sistemlerine Stack denilmektedir. Günlük hayatımızda stack biçiminde çalışan sistemlerle karşılaşmışızdır. Örneğin tabaklar üst üste konulduğunda son konulan önce alınır. Undo işlemi son yapılanı geri alır. Stack veri yapısı bazı tipik algoritmaları gerçekleştirmek için de kullanılabilmektedir. Stack sisteminde kuyruğa eleman eklemeye geleneksel olarak Push işlemi, kuyruktan eleman almaya da Pop işlemi denilmektedir:

```
public virtual void Push(Object obj)
public virtual Object Pop()
```

Yine Stack'teki eleman sayısı Count property'si ile elde edilmektedir. Stack de atipik bir collection sınıftır. Doğrudan ICollection arayüzünü desteklemektedir. Örneğin:

```
using System;
using System.Collections;

namespace CSD
{
classApp
    {
publicstaticvoid Main()
        {
Stack s = newStack();

for (int i = 0; i < 10; ++i)
                s.Push(i);

while (s.Count > 0)
            {
int val = (int)s.Pop();
Console.Write("{0} ", val);
            }
Console.WriteLine();
        }
    }
}
```

Stack sınıfının Stack<T> biçiminde generic versiyonu da vardır. Örneğin:

```
using System;
using System.Collections.Generic;

namespace CSD
{
classApp
    {
publicstaticvoid Main()
        {
Stack<int> s = newStack<int>();
```

```
for (int i = 0; i < 10; ++i)
                s.Push(i);

while (s.Count > 0)
            {
int val = s.Pop();
Console.Write("{0} ", val);
            }
Console.WriteLine();
        }
    }
}
```

Sınıf Çalışması: Klavyeden bri yazı okuyunuz. Sonra bu yazının karakterlerini tek tek bir Stack nesnesine Push ediniz. Sonra da tüm karakterleri Pop ederek yan yana yazdırınız.

Çözüm:

```
using System;
using System.Collections.Generic;

namespace CSD
{
classApp
    {
publicstaticvoid Main()
        {
Stack<char> s = newStack<char>();
string str;

Console.Write("Bir yazı giriniz:");
            str = Console.ReadLine();

foreach (char ch in str)
                s.Push(ch);
while (s.Count > 0)
Console.Write(s.Pop());
Console.WriteLine();
        }
    }
}
```

Referanslarda Eşitlik Karşılaştırmaları

C#'ta iki referans == ve != operatörleriyle karşılaştırılmak istendiğinde önce referanslara ilişkin sınıflarda bu karşılaştırmayı yapabilecek operatör metodu var mı diye bakılır. Varsa onlar çağrılır. Fakat böyle bir operatör metodu yoksa bu durum referanslar içerisindeki adreslerin karşılaştırılacağı anlamına gelir. Yani böylelikle iki referansın aynı nesneyi gösterip göstermediğine bakılacaktır. Ancak operatör metodu söz konusu olmadığında bizim == ve !=

operatörleriyle eşitlik karşılaştırması yapabilmemiz için iki referansın ya aynı sınıf türünden olması ya da aralarında türetme ilişkisi olması gerekir (yani biri diğerinin taban sınıfı olm alıdır). Biz operatör metodu söz konusu değilse >, >=, < ve <= operatörleriyle hiçbir durumda karşılaştırma yapamayız.

Örneğin biz iki string'i karşılaştırmak istediğimizde string sınıfının == ve != operatör metotları olduğu için onların içerisindeki adresleri değil, onların gösterdiği yerdeki nesne içerisindeki yazıların aynı olup olmadığını karşılaştırırız.

Bazen sınıfın == ve != operatör metotları olduğu durumda biz yine de adres karşılaştırması yapmak isteyebiliriz. Bunun için object sınıfının static ReferenceEquals metodu kullanılmaktadır. Örneğin:

```
using System;

namespace CSD
{
classApp
    {
publicstaticvoid Main()
        {
string s = "ankara";
string k = "an";

            k += "kara";

Console.WriteLine(s == k);                          // True
Console.WriteLine(object.ReferenceEquals(s, k));    // False
        }
    }
}
```

Anahtar Notlar: C# standartlarına göre taamamen aynı karakterlerden oluşan özdeş string'ler için her defasında ayrı bir string nesnesi oluşturulmaz. Aynı assemblideki özdeş string'ler için tek bir string nesnesi oluşturulur. Hepsine aynı nesne adresi atanır. Örneğin:

```
using System;

namespace CSD
{
classApp
    {
publicstaticvoid Main()
        {
string s = "ankara";
string k = "ankara";

Console.WriteLine(s == k);                          // True
Console.WriteLine(object.ReferenceEquals(s, k));    // True
        }
    }
}
```

Taban sınıfta == ve != operatör metotları bulunuyor olabilir. Eğer türemiş sınıfta bu metotlar yoksa taban sınıftaki metotlar devreye girer. Örneğin:

```
using System;

namespace CSD
{
classApp
    {
publicstaticvoid Main()
        {
B x = newB(10, 20);
B y = newB(10, 30);

Console.WriteLine(x == y);      // True
        }
    }

classA
    {
publicint ValA;

public A(int valA)
        {
            ValA = valA;
        }

publicstaticbooloperator ==(A x, A y)
        {
return x.ValA == y.ValA;
        }

publicstaticbooloperator !=(A x, A y)
        {
return x.ValA != y.ValA;
        }
    }

classB : A
    {
publicint ValB;

public B(int valA, int valB) : base(valA)
        {
            ValB = valB;
        }
    }
}
```

C#'ta aynı türden bile olsa iki yapı değişkeninin == ve != operatörleriyle karşılaştırılması için ilgili yapılarda bu operatör metotlarının bulunması gerekir.

Object sınıfının Equals Metotları

object sınıfının aşağıdaki gibi virtual bir Equals metodu vardır:

```
public virtual bool Equals(Object obj)
```

Bu metot int, long, double gibi temel türlere ilişkin yapılarda ve pek çok sınıfta override edilmiştir. Equals metodu bazı metotlar tarafından dolaylı olarak da çağrılmaktadır. Örneğin ArrayList sınıfının IList arayüzünden gelen Equals gibi, IndexOf gibi metotları collection içerisinde arama yaparken object sınıfından gelen bu sanal Equals metodunu çağırmaktadır. Bizim bu metotları sağlıklı olarak kullanabilmemiz için ilgili sınıfta bu metodu override etmemiz gerekir. Örneğin:

```
using System;
using System.Collections;

namespace CSD
{
classApp
    {
publicstaticvoid Main()
        {
ArrayList al = newArrayList();

for (int i = 0; i < 10; ++i)
                al.Add(newNumber(i));

Number x = newNumber(5);

if (al.Contains(x))
Console.WriteLine("Var");
else
Console.WriteLine("Yok");
        }
    }

classNumber
    {
privateint m_val;

public Number(int val)
        {
            m_val = val;
        }

publicoverridebool Equals(object obj)
        {
```

```
Number n = (Number)obj;

return n.m_val == m_val;
        }

publicint Val
        {
get { return m_val; }
        }
    }

}
```

ArrayList sınıfında Contains metodu muhtemelen aşağıdakine benzer biçimde yazılmıştır:

```
classMyArrayList
{
privateobject[] m_objs;
privateint m_count;
privateint m_capacity;

public MyArrayList()
    {
        m_objs = newobject[2];
        m_capacity = 2;
        m_count = 0;
    }

publicint Add(object o)
    {
if (m_count == m_capacity)
        {
            m_capacity *= 2;
object[] newObjs = newobject[m_capacity];
for (int i = 0; i < m_count; ++i)
                newObjs[i] = m_objs[i];
            m_objs = newObjs;
        }
        m_objs[m_count] = o;
        ++m_count;

return m_count - 1;
    }

publicint Count
    {
get { return m_count; }
    }

publicint Capacity
```

```
    {
get { return m_capacity; }
    }

publicbool Contains(object obj)
    {
for (int i = 0; i < m_count; ++i)
if (obj.Equals(m_objs[i]))
returntrue;
returnfalse;
    }

publicobjectthis[int index]
    {
get { return m_objs[index]; }
set { m_objs[index] = value; }
    }
   //...
}
```

Örneğin:

```
using System;
using System.Collections;

namespace CSD
{
classApp
    {
publicstaticvoid Main()
        {
ArrayList al = newArrayList();
int index;

for (int i = 0; i < 10; ++i)
                al.Add(newNumber(i));

Number x = newNumber(5);

if ((index = al.IndexOf(x)) == -1)
Console.WriteLine("Bulamadı");
else
Console.WriteLine("Buldu: {0}", index);
        }
    }

classNumber
    {
privateint m_val;
```

```
public Number(int val)
        {
            m_val = val;
        }

publicoverridebool Equals(object obj)
        {
Number n = (Number)obj;

return n.m_val == m_val;
        }

publicint Val
        {
get { return m_val; }
        }
    }

}
```

Eğer biz ilgili sınıfta Equals metodunu override etmezsek bu durumda object sınıfının Equals metodu çağrılır. O da referans eşitliğine bakmaktadır.

object sınıfında static bir Equals metodu daha vardır:

```
public static bool Equals(Object objA,Object objB)
```

Bu fonksiyon tatamen kendi içerisinde objA.Equals(objB) işlemini yapmaktadır. Yani:

```
object.Equals(a, b)
```

ile,

```
a.Equals(b)
```

tamamen işlevsel olarak eşdeğerdir. Başka bir anlatımla object sınıfının static Equals metodu şöle yazılmıştır:

```
classObject
{
//...
publicstaticbool Equals(object a, object b)
      {
return a.Equals(b);
}
//...
    }
```