Aggregate

Bir dizi accumulator işlevi uygular.

Aggregate yöntemi, bir veri kümesindeki tüm elemanları sırayla belirlenen bir işleme tabi tutarak bir yığın değer elde etmek için 2 parametreli bir Lambda İfadesi ile beraber kullanılır. Lambda İfadesinin (2 parametreli isimsiz fonksiyon), 1. parametresi yığın değişkenin son değerini, 2. parametresi ise sıradaki elemanın referansını barındırır.

```
string sentence = "the quick brown fox jumps over the lazy dog";

            // Split the string into individual words.
            string[] words = sentence.Split(' ');

            // Prepend each word to the beginning of the
            // new sentence to reverse the word order.
            string reversed = words.Aggregate((workingSentence, next) =>
                                                next + " " +
workingSentence);

            Console.WriteLine(reversed);

            // This code produces the following output:
            //
            // dog lazy the over jumps fox brown quick the
```

Bir dizi accumulator işlevi uygular. Belirtilen çekirdek değerini ilk accumulator değeri olarak kullanılır.

```
int[] ints = { 4, 8, 8, 3, 9, 0, 7, 8, 2 };

            // Count the even numbers in the array, using a seed value of
0.
            int numEven = ints.Aggregate(0, (total, next) =>
                                            next % 2 == 0 ? total + 1 :
total);

            Console.WriteLine("The number of even integers is: {0}",
numEven);

            // This code produces the following output:
            //
            // The number of even integers is: 6
```

Bir dizi accumulator işlevi uygular. Belirtilen çekirdek değerini ilk accumulator değeri olarak kullanılır ve belirtilen işlevin sonuç değeri seçmek için kullanılır.

```
string[] fruits = { "apple", "mango", "orange", "passionfruit", "grape" };

            // Determine whether any string in the array is longer than
"banana".
            string longestName =
                fruits.Aggregate("banana",
                                 (longest, next) =>
                                     next.Length > longest.Length ? next :
longest,
                // Return the final result as an upper case string.
                                 fruit => fruit.ToUpper());

            Console.WriteLine(
                "The fruit with the longest name is {0}.",
                longestName);

            // This code produces the following output:
            //
            // The fruit with the longest name is PASSIONFRUIT.
```

All : Tüm elemanların belirli şartı sağlayamadığını sağlamadığını kontrol eder.

```
class Pet
            {
                public string Name { get; set; }
                public int Age { get; set; }
            }

            public static void AllEx()
            {
                // Create an array of Pets.
                Pet[] pets = { new Pet { Name="Barley", Age=10 },
                               new Pet { Name="Boots", Age=4 },
                               new Pet { Name="Whiskers", Age=6 } };

                // Determine whether all pet names
                // in the array start with 'B'.
                bool allStartWithB = pets.All(pet =>

pet.Name.StartsWith("B"));

                Console.WriteLine(
                    "{0} pet names start with 'B'.",
                    allStartWithB ? "All" : "Not all");
            }
```

```
            // This code produces the following output:
            //
            //  Not all pet names start with 'B'.
```

Any

Kümede herhangi bir eleman olup olmadığını kontrol eder. Bir sıra herhangi bir öğe içerip içermediğini belirler.

```
  List<int> numbers = new List<int> { 1, 2 };
                bool hasElements = numbers.Any();

                Console.WriteLine("The list {0} empty.",
                    hasElements ? "is not" : "is");

                // This code produces the following output:
                //
                // The list is not empty.
```

Şartın herhangi bir eleman tarafından sağlanıp sağlanmadığını kontrol eder. Herhangi bir öğenin bir dizi koşulu karşılayıp karşılamadığını belirler.

```
class Pet
            {
                public string Name { get; set; }
                public int Age { get; set; }
                public bool Vaccinated { get; set; }
            }

            public static void AnyEx3()
            {
                // Create an array of Pets.
                Pet[] pets =
                    { new Pet { Name="Barley", Age=8, Vaccinated=true },
                      new Pet { Name="Boots", Age=4, Vaccinated=false },
                      new Pet { Name="Whiskers", Age=1, Vaccinated=false }
};

                // Determine whether any pets over age 1 are also
unvaccinated.
                bool unvaccinated =
                    pets.Any(p => p.Age > 1 && p.Vaccinated == false);

                Console.WriteLine(
                    "There {0} unvaccinated animals over age one.",
                    unvaccinated ? "are" : "are not any");
            }

            // This code produces the following output:
            //
```

```
            //  There are unvaccinated animals over age one.
```

Average : Sayısal değerler dizisi ortalamasını hesaplar.

```
long?[] longs = { null, 10007L, 37L, 399846234235L };

            double? average = longs.Average();

            Console.WriteLine("The average is {0}.", average);

            // This code produces the following output:
            //
            // The average is 133282081426.333.
```

Concat : İki diziyi birleştirir. Aynı elemanlar varsa birden fazla oluşur.

```
class Pet
            {
                public string Name { get; set; }
                public int Age { get; set; }
            }

            static Pet[] GetCats()
            {
                Pet[] cats = { new Pet { Name="Barley", Age=8 },
                               new Pet { Name="Boots", Age=4 },
                               new Pet { Name="Whiskers", Age=1 } };
                return cats;
            }

            static Pet[] GetDogs()
            {
                Pet[] dogs = { new Pet { Name="Bounder", Age=3 },
                               new Pet { Name="Snoopy", Age=14 },
                               new Pet { Name="Fido", Age=9 } };
                return dogs;
            }

            public static void ConcatEx1()
            {
                Pet[] cats = GetCats();
                Pet[] dogs = GetDogs();

                IEnumerable<string> query =
                    cats.Select(cat => cat.Name).Concat(dogs.Select(dog =>
dog.Name));

                foreach (string name in query)
                {
                    Console.WriteLine(name);
```

```
            }
        }

        // This code produces the following output:
        //
        // Barley
        // Boots
        // Whiskers
        // Bounder
        // Snoopy
        // Fido
```

Contains : Br dizinin belirtilen öğe içerip içermediğini belirler.

```
string[] fruits = { "apple", "banana", "mango", "orange", "passionfruit",
"grape" };

        string fruit = "mango";

        bool hasMango = fruits.Contains(fruit);

        Console.WriteLine(
            "The array {0} contain '{1}'.",
            hasMango ? "does" : "does not",
            fruit);

        // This code produces the following output:
        //
        // The array does contain 'mango'.
```

Count : Bir dizideki öğelerin sayısını döndürür.

```
string[] fruits = { "apple", "banana", "mango", "orange", "passionfruit",
"grape" };

        try
        {
            int numberOfFruits = fruits.Count();
            Console.WriteLine(
                "There are {0} fruits in the collection.",
                numberOfFruits);

        }
        catch (OverflowException)
        {
            Console.WriteLine("The count is too large to store as an
Int32.");
            Console.WriteLine("Try using the LongCount() method
instead.");
        }
```

```
            // This code produces the following output:
            //
            // There are 6 fruits in the collection.
```

DefaultIfEmpty : Boş ise varsayılanı getir. Sıra boş ise, aynı cinsten tek adet koleksiyonunda öğelerini belirtilen sıra veya tür parametresinin varsayılan değeri döndürür.

```
class Pet
            {
                public string Name { get; set; }
                public int Age { get; set; }
            }

            public static void DefaultIfEmptyEx1()
            {
                List<Pet> pets =
                    new List<Pet>{ new Pet { Name="Barley", Age=8 },
                                   new Pet { Name="Boots", Age=4 },
                                   new Pet { Name="Whiskers", Age=1 } };

                foreach (Pet pet in pets.DefaultIfEmpty())
                {
                    Console.WriteLine(pet.Name);
                }
            }

            /*
             This code produces the following output:

             Barley
             Boots
             Whiskers
            */
```

Distinct : Aynı elemanları içermeyen bir dizi döndürür.

```
            List<int> ages = new List<int> { 21, 46, 46, 55, 17, 21, 55, 55
};

            IEnumerable<int> distinctAges = ages.Distinct();

            Console.WriteLine("Distinct ages:");

            foreach (int age in distinctAges)
            {
                Console.WriteLine(age);
```

```
            }

            /*
             This code produces the following output:

             Distinct ages:
             21
             46
             55
             17
            */
```

Empty : Boş bir döner IEnumerable<T> , belirtilen türde bağımsız değişkene sahiptir.

```
string[] names1 = { "Hartono, Tommy" };
            string[] names2 = { "Adams, Terry", "Andersen, Henriette
Thaulow",
                                "Hedlund, Magnus", "Ito, Shu" };
            string[] names3 = { "Solanki, Ajay", "Hoeing, Helge",
                                "Andersen, Henriette Thaulow",
                                "Potra, Cristina", "Iallo, Lucio" };

            List<string[]> namesList =
                new List<string[]> { names1, names2, names3 };

            // Only include arrays that have four or more elements
            IEnumerable<string> allNames =
                namesList.Aggregate(Enumerable.Empty<string>(),
                (current, next) => next.Length > 3 ? current.Union(next) :
current);

            foreach (string name in allNames)
            {
                Console.WriteLine(name);
            }

            /*
             This code produces the following output:

             Adams, Terry
             Andersen, Henriette Thaulow
             Hedlund, Magnus
             Ito, Shu
             Solanki, Ajay
             Hoeing, Helge
             Potra, Cristina
             Iallo, Lucio
            */
```

Except : İki diziyi kümesi farkının değerleri karşılaştırmak için varsayılan eşitlik karşılaştırıcıyı kullanarak üretir.

```
double[] numbers1 = { 2.0, 2.1, 2.2, 2.3, 2.4, 2.5 };
            double[] numbers2 = { 2.2 };

            IEnumerable<double> onlyInFirstSet = numbers1.Except(numbers2);

            foreach (double number in onlyInFirstSet)
                Console.WriteLine(number);

            /*
             This code produces the following output:

             2
             2.1
             2.3
             2.4
             2.5
            */
```

First : Belirtilen bir koşula uygun bir dizideki ilk öğeyi döner.

```
int[] numbers = { 9, 34, 65, 92, 87, 435, 3, 54,
                                83, 23, 87, 435, 67, 12, 19 };

            int first = numbers.First(number => number > 80);

            Console.WriteLine(first);

            /*
             This code produces the following output:

             92
            */
```

FirstOrDefault : Öyle bir öğe bulunursa karşılayan bir koşul veya varsayılan değeri dizinin ilk öğesini döner.

```
string[] names = { "Hartono, Tommy", "Adams, Terry",
                                "Andersen, Henriette Thaulow",
                                "Hedlund, Magnus", "Ito, Shu" };

            string firstLongName = names.FirstOrDefault(name => name.Length
> 20);

            Console.WriteLine("The first long name is '{0}'.",
firstLongName);
```

```
            string firstVeryLongName = names.FirstOrDefault(name =>
name.Length > 30);

            Console.WriteLine(
                "There is {0} name longer than 30 characters.",
                string.IsNullOrEmpty(firstVeryLongName) ? "not a" : "a");

            /*
             This code produces the following output:

             The first long name is 'Andersen, Henriette Thaulow'.
             There is not a name longer than 30 characters.
            */
```

GroupBy : Gruplandırma işlemi

```
class Pet
            {
                public string Name { get; set; }
                public int Age { get; set; }
            }

            // Uses method-based query syntax.
            public static void GroupByEx1()
            {
                // Create a list of pets.
                List<Pet> pets =
                    new List<Pet>{ new Pet { Name="Barley", Age=8 },
                                   new Pet { Name="Boots", Age=4 },
                                   new Pet { Name="Whiskers", Age=1 },
                                   new Pet { Name="Daisy", Age=4 } };

                // Group the pets using Age as the key value
                // and selecting only the pet's Name for each value.
                IEnumerable<IGrouping<int, string>> query =
                    pets.GroupBy(pet => pet.Age, pet => pet.Name);

                // Iterate over each IGrouping in the collection.
                foreach (IGrouping<int, string> petGroup in query)
                {
                    // Print the key value of the IGrouping.
                    Console.WriteLine(petGroup.Key);
                    // Iterate over each value in the
                    // IGrouping and print the value.
                    foreach (string name in petGroup)
                        Console.WriteLine("  {0}", name);
                }
            }

            /*
```

```
             This code produces the following output:

             8
               Barley
             4
               Boots
               Daisy
             1
               Whiskers
            */
```

```
class Pet
            {
                public string Name { get; set; }
                public double Age { get; set; }
            }

            public static void GroupByEx3()
            {
                // Create a list of pets.
                List<Pet> petsList =
                    new List<Pet>{ new Pet { Name="Barley", Age=8.3 },
                                   new Pet { Name="Boots", Age=4.9 },
                                   new Pet { Name="Whiskers", Age=1.5 },
                                   new Pet { Name="Daisy", Age=4.3 } };

                // Group Pet objects by the Math.Floor of their age.
                // Then project an anonymous type from each group
                // that consists of the key, the count of the group's
                // elements, and the minimum and maximum age in the group.
                var query = petsList.GroupBy(
                    pet => Math.Floor(pet.Age),
                    (age, pets) => new
                    {
                        Key = age,
                        Count = pets.Count(),
                        Min = pets.Min(pet => pet.Age),
                        Max = pets.Max(pet => pet.Age)
                    });

                // Iterate over each anonymous type.
                foreach (var result in query)
                {
                    Console.WriteLine("\nAge group: " + result.Key);
                    Console.WriteLine("Number of pets in this age group: "
+ result.Count);
                    Console.WriteLine("Minimum age: " + result.Min);
                    Console.WriteLine("Maximum age: " + result.Max);
                }

                /*  This code produces the following output:
```

```
                    Age group: 8
                    Number of pets in this age group: 1
                    Minimum age: 8.3
                    Maximum age: 8.3

                    Age group: 4
                    Number of pets in this age group: 2
                    Minimum age: 4.3
                    Maximum age: 4.9

                    Age group: 1
                    Number of pets in this age group: 1
                    Minimum age: 1.5
                    Maximum age: 1.5
                */
            }
```

```
class Pet
            {
                public string Name { get; set; }
                public double Age { get; set; }
            }

            public static void GroupByEx4()
            {
                // Create a list of pets.
                List<Pet> petsList =
                    new List<Pet>{ new Pet { Name="Barley", Age=8.3 },
                                   new Pet { Name="Boots", Age=4.9 },
                                   new Pet { Name="Whiskers", Age=1.5 },
                                   new Pet { Name="Daisy", Age=4.3 } };

                // Group Pet.Age values by the Math.Floor of the age.
                // Then project an anonymous type from each group
                // that consists of the key, the count of the group's
                // elements, and the minimum and maximum age in the group.
                var query = petsList.GroupBy(
                    pet => Math.Floor(pet.Age),
                    pet => pet.Age,
                    (baseAge, ages) => new
                    {
                        Key = baseAge,
                        Count = ages.Count(),
                        Min = ages.Min(),
                        Max = ages.Max()
                    });

                // Iterate over each anonymous type.
                foreach (var result in query)
                {
```

```
                    Console.WriteLine("\nAge group: " + result.Key);
                    Console.WriteLine("Number of pets in this age group: "
+ result.Count);
                    Console.WriteLine("Minimum age: " + result.Min);
                    Console.WriteLine("Maximum age: " + result.Max);
                }

                /*  This code produces the following output:

                    Age group: 8
                    Number of pets in this age group: 1
                    Minimum age: 8.3
                    Maximum age: 8.3

                    Age group: 4
                    Number of pets in this age group: 2
                    Minimum age: 4.3
                    Maximum age: 4.9

                    Age group: 1
                    Number of pets in this age group: 1
                    Minimum age: 1.5
                    Maximum age: 1.5
                */
            }
```

GroupJoin : iki diziyi anahtarları eşitlik bakımından temel öğelerini belirtilirler ve sonuçları gruplandırır. Varsayılan eşitlik karşılaştırıcı anahtarları karşılaştırmak için kullanılır.

```
class Person
            {
                public string Name { get; set; }
            }

            class Pet
            {
                public string Name { get; set; }
                public Person Owner { get; set; }
            }

            public static void GroupJoinEx1()
            {
                Person magnus = new Person { Name = "Hedlund, Magnus" };
                Person terry = new Person { Name = "Adams, Terry" };
                Person charlotte = new Person { Name = "Weiss, Charlotte"
};

                Pet barley = new Pet { Name = "Barley", Owner = terry };
```

```
                Pet boots = new Pet { Name = "Boots", Owner = terry };
                Pet whiskers = new Pet { Name = "Whiskers", Owner =
charlotte };
                Pet daisy = new Pet { Name = "Daisy", Owner = magnus };

                List<Person> people = new List<Person> { magnus, terry,
charlotte };
                List<Pet> pets = new List<Pet> { barley, boots, whiskers,
daisy };

                // Create a list where each element is an anonymous
                // type that contains a person's name and
                // a collection of names of the pets they own.
                var query =
                    people.GroupJoin(pets,
                                     person => person,
                                     pet => pet.Owner,
                                     (person, petCollection) =>
                                         new
                                         {
                                             OwnerName = person.Name,
                                             Pets =
petCollection.Select(pet => pet.Name)
                                         });

                foreach (var obj in query)
                {
                    // Output the owner's name.
                    Console.WriteLine("{0}:", obj.OwnerName);
                    // Output each of the owner's pet's names.
                    foreach (string pet in obj.Pets)
                    {
                        Console.WriteLine("  {0}", pet);
                    }
                }
            }

            /*
             This code produces the following output:

             Hedlund, Magnus:
               Daisy
             Adams, Terry:
               Barley
               Boots
             Weiss, Charlotte:
               Whiskers
            */
```

Intersect : Set iki dizinin kesişimini değerleri karşılaştırmak için varsayılan eşitlik karşılaştırıcıyı kullanarak üretir.

```
int[] id1 = { 44, 26, 92, 30, 71, 38 };
```

```
            int[] id2 = { 39, 59, 83, 47, 26, 4, 30 };

            IEnumerable<int> both = id1.Intersect(id2);

            foreach (int id in both)
                Console.WriteLine(id);

            /*
             This code produces the following output:

             26
             30
            */
```

Join

İki diziyi eşleşen ayrıntılarına dayalıdır öğelerini belirtilirler. Varsayılan eşitlik karşılaştırıcı anahtarları karşılaştırmak için kullanılır.

```
  class Person
            {
                public string Name { get; set; }
            }

            class Pet
            {
                public string Name { get; set; }
                public Person Owner { get; set; }
            }

            public static void JoinEx1()
            {
                Person magnus = new Person { Name = "Hedlund, Magnus" };
                Person terry = new Person { Name = "Adams, Terry" };
                Person charlotte = new Person { Name = "Weiss, Charlotte"
};

                Pet barley = new Pet { Name = "Barley", Owner = terry };
                Pet boots = new Pet { Name = "Boots", Owner = terry };
                Pet whiskers = new Pet { Name = "Whiskers", Owner =
charlotte };
                Pet daisy = new Pet { Name = "Daisy", Owner = magnus };

                List<Person> people = new List<Person> { magnus, terry,
charlotte };
                List<Pet> pets = new List<Pet> { barley, boots, whiskers,
daisy };

                // Create a list of Person-Pet pairs where
                // each element is an anonymous type that contains a
                // Pet's name and the name of the Person that owns the Pet.
                var query =
```

```
                    people.Join(pets,
                                person => person,
                                pet => pet.Owner,
                                (person, pet) =>
                                    new { OwnerName = person.Name, Pet =
pet.Name });

                foreach (var obj in query)
                {
                    Console.WriteLine(
                        "{0} - {1}",
                        obj.OwnerName,
                        obj.Pet);
                }
            }

            /*
             This code produces the following output:

             Hedlund, Magnus - Daisy
             Adams, Terry - Barley
             Adams, Terry - Boots
             Weiss, Charlotte - Whiskers
            */
```

Last : Son öğe belirtilen bir koşula uygun bir dizi döndürür.

```
int[] numbers = { 9, 34, 65, 92, 87, 435, 3, 54,
                                83, 23, 87, 67, 12, 19 };

            int last = numbers.Last(num => num > 80);

            Console.WriteLine(last);

            /*
             This code produces the following output:

             87
            */
```

LastOrDefault : Öyle bir öğe bulunursa karşılayan bir koşul veya varsayılan değeri bir dizi son öğeyi döndürür.

```
            double[] numbers = { 49.6, 52.3, 51.0, 49.4, 50.2, 48.3 };

            double last50 = numbers.LastOrDefault(n => Math.Round(n) ==
50.0);
```

```
            Console.WriteLine("The last number that rounds to 50 is {0}.",
last50);

            double last40 = numbers.LastOrDefault(n => Math.Round(n) ==
40.0);

            Console.WriteLine(
                "The last number that rounds to 40 is {0}.",
                last40 == 0.0 ? "<DOES NOT EXIST>" : last40.ToString());

            /*
             This code produces the following output:

             The last number that rounds to 50 is 50.2.
             The last number that rounds to 40 is <DOES NOT EXIST>.
            */
```

LongCount : Sonucu long tipinde veren Count. Normal Count sonucu integer ti,pinde veriyor.

```
string[] fruits = { "apple", "banana", "mango",
                                    "orange", "passionfruit", "grape" };

            long count = fruits.LongCount();

            Console.WriteLine("There are {0} fruits in the collection.",
count);

            /*
             This code produces the following output:

             There are 6 fruits in the collection.
            */
```

Max : Her öğenin bir dizi dönüştürme işlevini çağırır ve en büyük döner Single değer.

```
List<long> longs = new List<long> { 4294967296L, 466855135L, 81125L };

            long max = longs.Max();

            Console.WriteLine("The largest number is {0}.", max);

            /*
             This code produces the following output:

             The largest number is 4294967296.
            */
```

```
            class Pet
```

```
            {
                public string Name { get; set; }
                public int Age { get; set; }
            }

            public static void MaxEx4()
            {
                Pet[] pets = { new Pet { Name="Barley", Age=8 },
                               new Pet { Name="Boots", Age=4 },
                               new Pet { Name="Whiskers", Age=1 } };

                int max = pets.Max(pet => pet.Age + pet.Name.Length);

                Console.WriteLine(
                    "The maximum pet age plus name length is {0}.",
                    max);
            }

            /*
             This code produces the following output:

             The maximum pet age plus name length is 14.
            */
```

Min : Bir dizideki en küçük değeri verir Decimal değerleri.

```
double[] doubles = { 1.5E+104, 9E+103, -2E+103 };

            double min = doubles.Min();

            Console.WriteLine("The smallest number is {0}.", min);

            /*
             This code produces the following output:

             The smallest number is -2E+103.
            */
```

```
class Pet
            {
                public string Name { get; set; }
                public int Age { get; set; }
            }

            public static void MinEx4()
            {
                Pet[] pets = { new Pet { Name="Barley", Age=8 },
                               new Pet { Name="Boots", Age=4 },
                               new Pet { Name="Whiskers", Age=1 } };
```

```
            int min = pets.Min(pet => pet.Age);

            Console.WriteLine("The youngest animal is age {0}.", min);
        }

        /*
         This code produces the following output:

         The youngest animal is age 1.
        */
```

OfType : ArrayList gibi generic olmayan sınıflarda, her eleman bject türüne dönüştürülü saklanmaktadır. OfType dizinin elemanlarını parametre olarak verilen türdeki elemanları alır.

```
System.Collections.ArrayList fruits = new System.Collections.ArrayList(4);
        fruits.Add("Mango");
        fruits.Add("Orange");
        fruits.Add("Apple");
        fruits.Add(3.0);
        fruits.Add("Banana");

        // Apply OfType() to the ArrayList.
        IEnumerable<string> query1 = fruits.OfType<string>();

        Console.WriteLine("Elements of type 'string' are:");
        foreach (string fruit in query1)
        {
            Console.WriteLine(fruit);
        }

        // The following query shows that the standard query operators
such as
        // Where() can be applied to the ArrayList type after calling
OfType().
        IEnumerable<string> query2 =
            fruits.OfType<string>().Where(fruit =>
fruit.ToLower().Contains("n"));

        Console.WriteLine("\nThe following strings contain 'n':");
        foreach (string fruit in query2)
        {
            Console.WriteLine(fruit);
        }

        // This code produces the following output:
        //
        // Elements of type 'string' are:
        // Mango
        // Orange
        // Apple
        // Banana
```

```
            //
            // The following strings contain 'n':
            // Mango
            // Orange
            // Banana
```

OrderBy : Bir anahtara göre artan bir dizinin öğeleri sıralar.

```
  class Pet
            {
                public string Name { get; set; }
                public int Age { get; set; }
            }

            public static void OrderByEx1()
            {
                Pet[] pets = { new Pet { Name="Barley", Age=8 },
                               new Pet { Name="Boots", Age=4 },
                               new Pet { Name="Whiskers", Age=1 } };

                IEnumerable<Pet> query = pets.OrderBy(pet => pet.Age);

                foreach (Pet pet in query)
                {
                    Console.WriteLine("{0} - {1}", pet.Name, pet.Age);
                }
            }

            /*
             This code produces the following output:

             Whiskers - 1
             Boots - 4
             Barley - 8
            */
```

OrderByDescending : Bir anahtara göre azalan bir dizinin öğeleri sıralar.

```
            /// <summary>
            /// This IComparer class sorts by the fractional part of the
decimal number.
            /// </summary>
            public class SpecialComparer : IComparer<decimal>
            {
                /// <summary>
                /// Compare two decimal numbers by their fractional parts.
                /// </summary>
                /// <param name="d1">The first decimal to compare.</param>
                /// <param name="d2">The second decimal to compare.</param>
                /// <returns>1 if the first decimal's fractional part
                /// is greater than the second decimal's fractional part,
```

```
                /// -1 if the first decimal's fractional
                /// part is less than the second decimal's fractional part,
                /// or the result of calling Decimal.Compare()
                /// if the fractional parts are equal.</returns>
                public int Compare(decimal d1, decimal d2)
                {
                    decimal fractional1, fractional2;

                    // Get the fractional part of the first number.
                    try
                    {
                        fractional1 = decimal.Remainder(d1,
decimal.Floor(d1));
                    }
                    catch (DivideByZeroException)
                    {
                        fractional1 = d1;
                    }
                    // Get the fractional part of the second number.
                    try
                    {
                        fractional2 = decimal.Remainder(d2,
decimal.Floor(d2));
                    }
                    catch (DivideByZeroException)
                    {
                        fractional2 = d2;
                    }

                    if (fractional1 == fractional2)
                        return Decimal.Compare(d1, d2);
                    else if (fractional1 > fractional2)
                        return 1;
                    else
                        return -1;
                }
            }

            public static void OrderByDescendingEx1()
            {
                List<decimal> decimals =
                    new List<decimal> { 6.2m, 8.3m, 0.5m, 1.3m, 6.3m, 9.7m
};

                IEnumerable<decimal> query =
                    decimals.OrderByDescending(num =>
                                                num, new
SpecialComparer());

                foreach (decimal num in query)
                {
                    Console.WriteLine(num);
                }
```

```
            }

            /*
             This code produces the following output:

             9.7
             0.5
             8.3
             6.3
             1.3
             6.2
            */
```

Range : Tamsayı numaraları belirtilen aralık içinde bir dizi oluşturur.

```
// Generate a sequence of integers from 1 to 10
            // and then select their squares.
            IEnumerable<int> squares = Enumerable.Range(1, 10).Select(x =>
x * x);

            foreach (int num in squares)
            {
                Console.WriteLine(num);
            }

            /*
             This code produces the following output:

             1
             4
             9
             16
             25
             36
             49
             64
             81
             100
            */
```

Repeat : Bir yinelenen değer içeren bir dizi oluşturur.

```
            IEnumerable<string> strings =
                Enumerable.Repeat("I like programming.", 15);

            foreach (String str in strings)
            {
                Console.WriteLine(str);
            }
```

```
            /*
             This code produces the following output:

             I like programming.
             I like programming.
             I like programming.
             I like programming.
             I like programming.
             I like programming.
             I like programming.
             I like programming.
             I like programming.
             I like programming.
             I like programming.
             I like programming.
             I like programming.
             I like programming.
             I like programming.
            */
```

Reverse : Bir dizideki öğelerin sırasını ters çevirir.

```
char[] apple = { 'a', 'p', 'p', 'l', 'e' };

            char[] reversed = apple.Reverse().ToArray();

            foreach (char chr in reversed)
            {
                Console.Write(chr + " ");
            }
            Console.WriteLine();

            /*
             This code produces the following output:

             e l p p a
            */
```

Select : Yeni bir form içine bir dizinin her öğesi projeleri.

```
  IEnumerable<int> squares =
                Enumerable.Range(1, 10).Select(x => x * x);

            foreach (int num in squares)
            {
                Console.WriteLine(num);
            }
            /*
             This code produces the following output:
```

```
             1
             4
             9
             16
             25
             36
             49
             64
             81
             100
            */
```

```
string[] fruits = { "apple", "banana", "mango", "orange",
                                    "passionfruit", "grape" };

            var query =
                fruits.Select((fruit, index) =>
                                    new { index, str = fruit.Substring(0,
index) });

            foreach (var obj in query)
            {
                Console.WriteLine("{0}", obj);
            }

            /*
             This code produces the following output:

             {index=0, str=}
             {index=1, str=b}
             {index=2, str=ma}
             {index=3, str=ora}
             {index=4, str=pass}
             {index=5, str=grape}
            */
```

SelectMany : Her öğe için bir dizi projeleri bir IEnumerable<T> ve bir dizi elde edilen dizilere düzleştirir.

```
class PetOwner
            {
                public string Name { get; set; }
                public List<String> Pets { get; set; }
            }

            public static void SelectManyEx1()
            {
                PetOwner[] petOwners =
                    { new PetOwner { Name="Higa, Sidney",
```

```
                            Pets = new List<string>{ "Scruffy", "Sam" } },
                        new PetOwner { Name="Ashkenazi, Ronen",
                            Pets = new List<string>{ "Walker", "Sugar" } },
                        new PetOwner { Name="Price, Vernette",
                            Pets = new List<string>{ "Scratches", "Diesel" }
} };

                // Query using SelectMany().
                IEnumerable<string> query1 = petOwners.SelectMany(petOwner
=> petOwner.Pets);

                Console.WriteLine("Using SelectMany():");

                // Only one foreach loop is required to iterate
                // through the results since it is a
                // one-dimensional collection.
                foreach (string pet in query1)
                {
                    Console.WriteLine(pet);
                }

                // This code shows how to use Select()
                // instead of SelectMany().
                IEnumerable<List<String>> query2 =
                    petOwners.Select(petOwner => petOwner.Pets);

                Console.WriteLine("\nUsing Select():");

                // Notice that two foreach loops are required to
                // iterate through the results
                // because the query returns a collection of arrays.
                foreach (List<String> petList in query2)
                {
                    foreach (string pet in petList)
                    {
                        Console.WriteLine(pet);
                    }
                    Console.WriteLine();
                }
            }

            /*
             This code produces the following output:

             Using SelectMany():
             Scruffy
             Sam
             Walker
             Sugar
             Scratches
             Diesel

             Using Select():
```

```
             Scruffy
             Sam

             Walker
             Sugar

             Scratches
             Diesel
            */
```

SequenceEqual : Kendi türü için varsayılan eşitlik karşılaştırıcıyı kullanarak öğeleri karşılaştırarak iki diziyi eşit olup olmadığını belirler.

```
  class Pet
            {
                public string Name { get; set; }
                public int Age { get; set; }
            }

            public static void SequenceEqualEx1()
            {
                Pet pet1 = new Pet { Name = "Turbo", Age = 2 };
                Pet pet2 = new Pet { Name = "Peanut", Age = 8 };

                // Create two lists of pets.
                List<Pet> pets1 = new List<Pet> { pet1, pet2 };
                List<Pet> pets2 = new List<Pet> { pet1, pet2 };

                bool equal = pets1.SequenceEqual(pets2);

                Console.WriteLine(
                    "The lists {0} equal.",
                    equal ? "are" : "are not");
            }

            /*
             This code produces the following output:

             The lists are equal.
            */
```

Single : Tek öğe, belirtilen bir koşula uygun ve birden fazla öğe varsa, aykırı bir dizi döndürür.

```
string[] fruits = { "apple", "banana", "mango",
                                    "orange", "passionfruit", "grape" };

            string fruit1 = fruits.Single(fruit => fruit.Length > 10);

            Console.WriteLine(fruit1);
```

```
            /*
             This code produces the following output:

             passionfruit
            */
```

SingleOrDefault : Tek öğe böyle bir öğe yoksa karşılayan belirtilen bir koşul veya varsayılan değeri bir dizi döndürür; birden fazla öğe koşulu karşılamazsa, bu yöntem bir istisna atar.

```
string[] fruits = { "apple", "banana", "mango",
                                    "orange", "passionfruit", "grape" };

            string fruit1 = fruits.SingleOrDefault(fruit => fruit.Length >
10);

            Console.WriteLine(fruit1);

            /*
             This code produces the following output:

             passionfruit
            */
```

Skip : Verilen sayı kadar elemanı atlar. Öğeleri bir sıra içinde belirtilen sayıda atlar ve kalan öğeleri döndürür.

```
   int[] grades = { 59, 82, 70, 56, 92, 98, 85 };

            IEnumerable<int> lowerGrades =
                grades.OrderByDescending(g => g).Skip(3);

            Console.WriteLine("All grades except the top three are:");
            foreach (int grade in lowerGrades)
            {
                Console.WriteLine(grade);
            }

            /*
             This code produces the following output:

             All grades except the top three are:
             82
             70
             59
             56
```

```
            */
```

SkipWhile : Belirtilen bir koşul doğru ve geriye kalan öğeleri döndürür sürece bir sýradaki öðeler atlar. Öğenin dizin doğrulama işlevi mantığı içinde kullanılır.

```
            int[] amounts = { 5000, 2500, 9000, 8000,
                              6500, 4000, 1500, 5500 };

            IEnumerable<int> query =
                amounts.SkipWhile((amount, index) => amount > index *
1000);

            foreach (int amount in query)
            {
                Console.WriteLine(amount);
            }

            /*
             This code produces the following output:

             4000
             1500
             5500
            */
```

```
            int[] grades = { 59, 82, 70, 56, 92, 98, 85 };

            IEnumerable<int> lowerGrades =
                grades
                .OrderByDescending(grade => grade)
                .SkipWhile(grade => grade >= 80);

            Console.WriteLine("All grades below 80:");
            foreach (int grade in lowerGrades)
            {
                Console.WriteLine(grade);
            }

            /*
             This code produces the following output:

             All grades below 80:
             70
             59
             56
            */
```

Sum : Bir dizi toplamını hesaplar Decimal değerleri.

```
List<float> numbers = new List<float> { 43.68F, 1.25F, 583.7F, 6.5F };

            float sum = numbers.Sum();

            Console.WriteLine("The sum of the numbers is {0}.", sum);

            /*
             This code produces the following output:

             The sum of the numbers is 635.13.
            */
```

```
class Package
            {
                public string Company { get; set; }
                public double Weight { get; set; }
            }

            public static void SumEx1()
            {
                List<Package> packages =
                    new List<Package>
                        { new Package { Company = "Coho Vineyard", Weight =
25.2 },
                          new Package { Company = "Lucerne Publishing",
Weight = 18.7 },
                          new Package { Company = "Wingtip Toys", Weight =
6.0 },
                          new Package { Company = "Adventure Works", Weight
= 33.8 } };

                double totalWeight = packages.Sum(pkg => pkg.Weight);

                Console.WriteLine("The total weight of the packages is:
{0}", totalWeight);
            }

            /*
             This code produces the following output:

             The total weight of the packages is: 83.7
            */
```

Take : Bitişik öğeleri belirtilen sayıda bir dizi başından döndürür. Başından itibaren belirtilen sayıda elemanı alır.

```
int[] grades = { 59, 82, 70, 56, 92, 98, 85 };
```

```
            IEnumerable<int> topThreeGrades =
                grades.OrderByDescending(grade => grade).Take(3);

            Console.WriteLine("The top three grades are:");
            foreach (int grade in topThreeGrades)
            {
                Console.WriteLine(grade);
            }
            /*
             This code produces the following output:

             The top three grades are:
             98
             92
             85
            */
```

TakeWhile : Belirtilen bir koşul true olduğu sürece öğeleri bir dizi döner.

```
            string[] fruits = { "apple", "banana", "mango", "orange",
                                "passionfruit", "grape" };

            IEnumerable<string> query =
                fruits.TakeWhile(fruit => String.Compare("orange", fruit,
true) != 0);

            foreach (string fruit in query)
            {
                Console.WriteLine(fruit);
            }

            /*
             This code produces the following output:

             apple
             banana
             mango
            */
```

```
            string[] fruits = { "apple", "passionfruit", "banana", "mango",
                                "orange", "blueberry", "grape",
"strawberry" };

            IEnumerable<string> query =
                fruits.TakeWhile((fruit, index) => fruit.Length >= index);

            foreach (string fruit in query)
            {
                Console.WriteLine(fruit);
```

```
            }

            /*
             This code produces the following output:

             apple
             passionfruit
             banana
             mango
             orange
             blueberry
            */
```

ThenBy : Bir sonraki öğelerin bir anahtara göre artan sırada sıralama gerçekleştirir.

```
            string[] fruits = { "grape", "passionfruit", "banana", "mango",
                                "orange", "raspberry", "apple",
"blueberry" };

            // Sort the strings first by their length and then
            //alphabetically by passing the identity selector function.
            IEnumerable<string> query =
                fruits.OrderBy(fruit => fruit.Length).ThenBy(fruit =>
fruit);

            foreach (string fruit in query)
            {
                Console.WriteLine(fruit);
            }

            /*
                This code produces the following output:

                apple
                grape
                mango
                banana
                orange
                blueberry
                raspberry
                passionfruit
            */
```

ThenByDescending : Bir sonraki tüm öğeleri, azalan sırada sıralama anahtarına göre gerçekleştirir.

```
  public class CaseInsensitiveComparer : IComparer<string>
            {
                public int Compare(string x, string y)
                {
                    return string.Compare(x, y, true);
                }
```

```
            }

            public static void ThenByDescendingEx1()
            {
                string[] fruits = { "apPLe", "baNanA", "apple", "APple",
"orange", "BAnana", "ORANGE", "apPLE" };

                // Sort the strings first ascending by their length and
                // then descending using a custom case insensitive
comparer.
                IEnumerable<string> query =
                    fruits
                    .OrderBy(fruit => fruit.Length)
                    .ThenByDescending(fruit => fruit, new
CaseInsensitiveComparer());

                foreach (string fruit in query)
                {
                    Console.WriteLine(fruit);
                }
            }

            /*
                This code produces the following output:

                apPLe
                apple
                APple
                apPLE
                orange
                ORANGE
                baNanA
                BAnana
            */
```

ToArray : Sonuç diziye aktarılır.

```
class Package
            {
                public string Company { get; set; }
                public double Weight { get; set; }
            }

            public static void ToArrayEx1()
            {
                List<Package> packages =
                    new List<Package>
                        { new Package { Company = "Coho Vineyard", Weight =
25.2 },
```

```
                            new Package { Company = "Lucerne Publishing",
Weight = 18.7 },
                            new Package { Company = "Wingtip Toys", Weight =
6.0 },
                            new Package { Company = "Adventure Works", Weight
= 33.8 } };

                string[] companies = packages.Select(pkg =>
pkg.Company).ToArray();

                foreach (string company in companies)
                {
                    Console.WriteLine(company);
                }
            }

            /*
             This code produces the following output:

             Coho Vineyard
             Lucerne Publishing
             Wingtip Toys
             Adventure Works
            */
```

ToDictionary : Sonuçlarıu anahtar, değer çifti şeklinde döndürür.

```
  class Package
            {
                public string Company { get; set; }
                public double Weight { get; set; }
                public long TrackingNumber { get; set; }
            }

            public static void ToDictionaryEx1()
            {
                List<Package> packages =
                    new List<Package>
                        { new Package { Company = "Coho Vineyard", Weight =
25.2, TrackingNumber = 89453312L },
                            new Package { Company = "Lucerne Publishing",
Weight = 18.7, TrackingNumber = 89112755L },
                            new Package { Company = "Wingtip Toys", Weight =
6.0, TrackingNumber = 299456122L },
                            new Package { Company = "Adventure Works", Weight
= 33.8, TrackingNumber = 4665518773L } };

                // Create a Dictionary of Package objects,
                // using TrackingNumber as the key.
                Dictionary<long, Package> dictionary =
                    packages.ToDictionary(p => p.TrackingNumber);
```

```
                foreach (KeyValuePair<long, Package> kvp in dictionary)
                {
                    Console.WriteLine(
                        "Key {0}: {1}, {2} pounds",
                        kvp.Key,
                        kvp.Value.Company,
                        kvp.Value.Weight);
                }
            }

            /*
             This code produces the following output:

             Key 89453312: Coho Vineyard, 25.2 pounds
             Key 89112755: Lucerne Publishing, 18.7 pounds
             Key 299456122: Wingtip Toys, 6 pounds
             Key 4665518773: Adventure Works, 33.8 pounds
            */
```

ToList : Sonuçları List sınıfına aktarır.

```
          string[] fruits = { "apple", "passionfruit", "banana", "mango",
                                    "orange", "blueberry", "grape",
"strawberry" };

            List<int> lengths = fruits.Select(fruit =>
fruit.Length).ToList();

            foreach (int length in lengths)
            {
                Console.WriteLine(length);
            }

            /*
             This code produces the following output:

             5
             12
             6
             5
             6
             9
             5
             10
            */
```

ToLookup : Belirli ortak özelliğe sahip veya isteğimize göre gruplandırdığımız nesnelere bir tek nesne kullanarak erişmek için kullanılır.

```
   class Package
            {
                public string Company { get; set; }
                public double Weight { get; set; }
                public long TrackingNumber { get; set; }
            }

            public static void ToLookupEx1()
            {
                // Create a list of Packages.
                List<Package> packages =
                    new List<Package>
                        { new Package { Company = "Coho Vineyard",
                              Weight = 25.2, TrackingNumber = 89453312L },
                          new Package { Company = "Lucerne Publishing",
                              Weight = 18.7, TrackingNumber = 89112755L },
                          new Package { Company = "Wingtip Toys",
                              Weight = 6.0, TrackingNumber = 299456122L },
                          new Package { Company = "Contoso
Pharmaceuticals",
                              Weight = 9.3, TrackingNumber = 670053128L },
                          new Package { Company = "Wide World Importers",
                              Weight = 33.8, TrackingNumber = 4665518773L }
};

                // Create a Lookup to organize the packages.
                // Use the first character of Company as the key value.
                // Select Company appended to TrackingNumber
                // as the element values of the Lookup.
                ILookup<char, string> lookup =
                    packages
                    .ToLookup(p => Convert.ToChar(p.Company.Substring(0,
1)),
                              p => p.Company + " " + p.TrackingNumber);

                // Iterate through each IGrouping in the Lookup.
                foreach (IGrouping<char, string> packageGroup in lookup)
                {
                    // Print the key value of the IGrouping.
                    Console.WriteLine(packageGroup.Key);
                    // Iterate through each value in the
                    // IGrouping and print its value.
                    foreach (string str in packageGroup)
                        Console.WriteLine("    {0}", str);
                }
            }

            /*
             This code produces the following output:

             C
                 Coho Vineyard 89453312
                 Contoso Pharmaceuticals 670053128
```

```
             L
                 Lucerne Publishing 89112755
             W
                 Wingtip Toys 299456122
                 Wide World Importers 4665518773
            */
```

Union : İki diziyi kümesinin birleşimini varsayılan eşitlik karşılaştırıcıyı kullanarak üretir.

```
  int[] ints1 = { 5, 3, 9, 7, 5, 9, 3, 7 };
            int[] ints2 = { 8, 3, 6, 4, 4, 9, 1, 0 };

            IEnumerable<int> union = ints1.Union(ints2);

            foreach (int num in union)
            {
                Console.Write("{0} ", num);
            }

            /*
             This code produces the following output:

             5 3 9 7 8 6 4 1 0
            */
```

Where : Bir dizi koşul üzerinde temel değerleri filtre uygular.

```
  List<string> fruits =
                new List<string> { "apple", "passionfruit", "banana",
"mango",
                                   "orange", "blueberry", "grape",
"strawberry" };

            IEnumerable<string> query = fruits.Where(fruit => fruit.Length
< 6);

            foreach (string fruit in query)
            {
                Console.WriteLine(fruit);
            }
            /*
             This code produces the following output:

             apple
             mango
             grape
            */
```

```
   int[] numbers = { 0, 30, 20, 15, 90, 85, 40, 75 };

            IEnumerable<int> query =
                numbers.Where((number, index) => number <= index * 10);

            foreach (int number in query)
            {
                Console.WriteLine(number);
            }
            /*
             This code produces the following output:

             0
             20
             15
             40
            */
```

Zip :

Belirtilen işlev bir dizi sonuçları üreten iki diziyi öğelere uygulanır. Her iki kümenin elemanlarını alıp belirlene fonksiyona iletir. Bu sayaede ilişkili iki küme üzerinde işlemler yapılabilir. Her 2 kümenin eleman sayısının eşit olmasına gerek oktur. En düşük eleman sayısı olan küme kadar döngü işletilmektedir.

```
         int[] numbers = { 1, 2, 3, 4 };
            string[] words = { "one", "two", "three" };

            var numbersAndWords = numbers.Zip(words, (first, second) =>
first + " " + second);

            foreach (var item in numbersAndWords)
                Console.WriteLine(item);

            // This code produces the following output:

            // 1 one
            // 2 two
            // 3 three
```
