public class Person

{

private string firstName, middleName, lastName;

public Person(string firstName, string middleName, string lastName)

{

this.firstName = firstName;

this.middleName = middleName;

this.lastName = lastName;

}

public IEnumerable<string> Names

{

get

{

yield return firstName;

yield return middleName;

yield return lastName;

}

}

}

var n = new Person("Sam", "Edward", "Hill");

foreach( var nm in n.Names)

{

Console.WriteLine("{0}", nm);

}

var list = new ArrayList();

list.Add(1);

list.Add(2);

list.Add(3);

// this won't work

//Console.WriteLine(list.Select(i => (int)i).Sum());

Console.WriteLine(list.Cast<int>().Sum());

// cast not valid

//Console.WriteLine(list.Cast<float>().Average());

//

var arr2 = new[]{1,2,3};

var arre = arr2.AsEnumerable();

// simply casts to IEnumerable<int>

ParallelQuery<int> pq =

arr2.AsParallel();

// AsQueryable()

var numbers = Enumerable.Range(1,4);

var squares = numbers.Select(x => x\*x);

Console.WriteLine(squares);

string sentence = "This is a nice sentence";

var wordLengths =

sentence.Split().Select(w => w.Length);

Console.WriteLine(wordLengths);

// keep both word and length

var wordsWithLength =

sentence.Split().Select(w => new { Word = w, Size = w.Length});

Console.WriteLine(wordsWithLength);

// get all the words

var sequences = new[]{ "red,green,blue", "orange", "white,pink" };

**// use Select**

var allWords = sequences.SelectMany(s => s.Split(',')); // select

Console.WriteLine(allWords);

// select each pair (cross product) from two collections

string [] objects = { "house", "car", "bicycle" };

string [] colors = { "red", "green", "gray" };

var pairs = colors.SelectMany(z => objects, (c, o) => $"{c} {o}");

Console.WriteLine(pairs);

var numbers = Enumerable.Range(1,10);

var evenNumbers = numbers.Where(n => n%2 == 0);

Console.WriteLine(evenNumbers);

// combine projection and filtering

var oddSquares = numbers.Select(x => x\*x).Where(y => y%2 != 0);

Console.WriteLine(oddSquares);

// filter by type

object [] values = { 1, 2.5, 3, 4.56 };

var wholeNumbers = values.OfType<int>(); // try float or double

Console.WriteLine(wholeNumbers);

public class Exercise1

{

public static IEnumerable<int> myFilter(IEnumerable<int> input)

{

var ret = input.Where( n => n%2 == 0 ).Select( y => y \* y ).Where( z => z<=50);

return ret;

}

}

Exercise.myFilter(Enumerable.Range(1, 10));

void Main()

{

var rand = new Random();

var randomValues = Enumerable.Range(1,10).Select(\_ => rand.Next(10) - 5);

var csvString = new Func<IEnumerable<int>,string>(values =>

{

return string.Join(",", values.Select(v => v.ToString()).ToArray());

});

// different set of values each time

Console.WriteLine(csvString(randomValues));

Console.WriteLine(csvString(randomValues.OrderBy(x => x)));

Console.WriteLine(csvString(randomValues.OrderByDescending(x => x)));

var people = new List<Person>

{

new Person{ Name = "Adam", Age = 36 },

new Person { Name = "Boris", Age = 18 },

new Person { Name = "Claire", Age = 36 },

new Person { Name = "Adam", Age = 20 },

new Person { Name = "Jack", Age = 20 }

};

Console.WriteLine(people);

Console.WriteLine(people.OrderBy(p => p.Name));

// result is IOrderedEnumerable<Person>

Console.WriteLine(people.OrderBy(p => p.Age)

.ThenByDescending(p => p.Name));

// reverse a string

string s = "This is a test";

Console.WriteLine(new string(s.Reverse().ToArray()));

}

// Define other methods and classes here

class Person

{

public string Name;

public int Age;

}

void Main()

{

var people = new List<Person>{

new Person { Name = "Adam", Age = 20 },

new Person{ Name = "Adam", Age = 36 },

new Person { Name = "Boris", Age = 18 },

new Person { Name = "Claire", Age = 36 },

new Person { Name = "Adam", Age = 20 }, // dup

new Person { Name = "Jack", Age = 20 }

};

IEnumerable<IGrouping<string, Person>> byName = people.GroupBy(p => p.Name);

//Console.WriteLine(byName);

// IEnumerable<IGrouping<bool, Person>>

Console.WriteLine(people.GroupBy(p => p.Age < 30));

// same, but we only keep the values instead of

// entire objects

var byAgeNames = people.GroupBy(p => p.Age, p => p.Name);

//Console.WriteLine(byAgeNames);

foreach (var item in byAgeNames)

{

Console.WriteLine($"These people are {item.Key} years old");

foreach (var name in item)

Console.WriteLine($" - {name}");

}

}

class Person

{

public string Name;

public int Age;

}

var word1 = "helloooo";

var word2 = "help";

// distinct letters in helloooo

Console.WriteLine(word1.Distinct());

// letters in both word1 and word2

var lettersInBoth = word1.Intersect(word2);

Console.WriteLine(lettersInBoth);

// letters in all words

Console.WriteLine(word1.Union(word2));

// letters in word1, but not in word1

Console.WriteLine(word1.Except(word2));

public class Exercise2

{

public static IEnumerable<int> merge(IEnumerable<int> a, IEnumerable<int> b)

{

var uniqueInOne = a.Except(b);

var uniqueInTwo = b.Except(a);

var combined = uniqueInOne.Union(uniqueInTwo);

return combined;

}

}

var ans = Exercise2.merge(

new int[] { 1, 2, 3, 4, 5 },

new int[] { 1, 3, 5, 7, 9 });

int [] numbers = { 1, 2, 3, 4, 5};

Console.WriteLine("Are all numbers > 0? " +

numbers.All(x => x > 0));

Console.WriteLine("Are all numbers odd? " +

numbers.All(x => x%2 == 1));

// once the iterator finds a match, it doesn't check other elements

Console.WriteLine("Any number less than two? " + numbers.Any(x => x < 2));

// also, Any() is a way to check whether a collection is empty

Console.WriteLine(new int[]{/\*42\*/}.Any());

Console.WriteLine("Contains 5? " + numbers.Contains(5)); // not a lambda here!

Console.WriteLine("Number of odd elements: " + numbers.Count(x => x % 2 == 1));

Console.WriteLine("Total no. of elements: " + numbers.Count()); // beware!

var numbers = new[]{ 3, 3, 2, 2, 1, 1, 2, 2, 3, 3 };

Console.WriteLine(numbers.Skip(2).Take(6));

Console.WriteLine(new int[]{}.Take(2));

Console.WriteLine(numbers.SkipWhile(i => i == 3));

Console.WriteLine(numbers.TakeWhile(i => i > 1));

public static int LengthOfPositive(IEnumerable<int> input)

{

var skipStartingNegatives = input.SkipWhile(i => i < 0);

var getNextPositives = skipStartingNegatives.TakeWhile(i => i > 0);

return getNextPositives.Count();

}

**int cnt = Exercise3.LengthOfPositive(new int[] { -3, -1, 3, 7, 1, -3, 7 });**