Babeș-Bolyai Universität
Cluj Napoca
Fakultät für Mathematik und Informatik
Fortgeschrittene Programmierungsmethoden

Seminar 12

1. LINQ Tutorial

LINQ ist eine Menge von Technologien, die die direkte Integration von Abfragefunktionen in die C # -Sprache unterstützen. Solche Abfragen werden normalerweise als einfache Zeichenfolgen ohne Typprüfung zur Kompilierungszeit und ohne IntelliSense-Unterstützung ausgedrückt.

Möglicherweise möchte man Daten aus verschiedenen Quellen abfragen, z. B. Arrays, Wörterbücher, XML-Dateien und Entitäten, die aus dem Entity-Framework erstellt wurden.

Anstatt für jede Datenquelle eine andere API verwenden zu müssen, bietet LINQ ein konsistentes und einheitliches Programmiermodell für die Arbeit mit allen unterstützten Datenquellen.

Einige der am häufigsten verwendeten LINQ-Datenquellen, die alle Teil des .NET-Frameworks sind, sind:

- LINQ to Objects: Für IEnumerable Collections, z. B. Dictionary und List
- LINQ to Entities: für Entity Framework im Objektkontext
- LINQ to XML: für XML-Dokumente

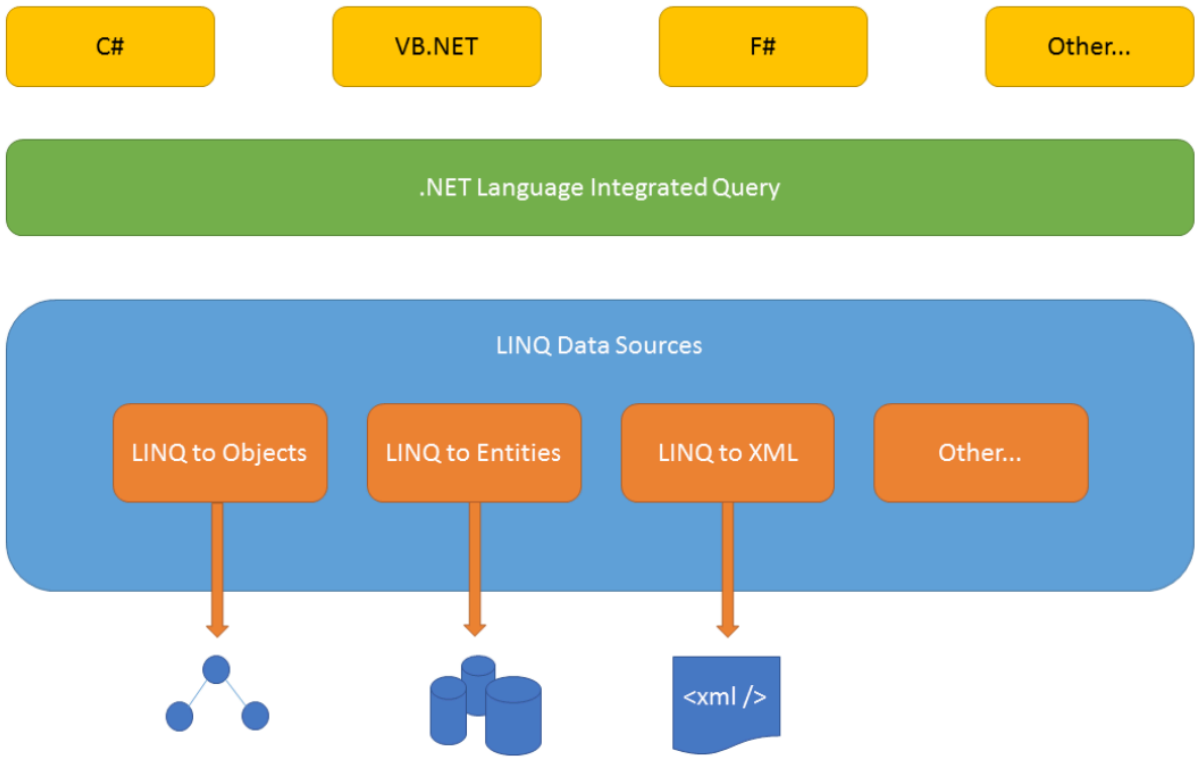
Solange der Namespace System.Linq im Code enthalten ist, können Daten aus allen oben genannten Datenquellen abgefragt werden.

Übersicht über Abfrageausdrücke (msdn doc)

- Abfrageausdrücke können verwendet werden, um Daten aus einer beliebigen LINQ-fähigen Datenquelle abzufragen und zu transformieren. Mit einer einzigen Abfrage können z.B. Daten aus einer SQL-Datenbank abgerufen und ein XML-Stream als Ausgabe generiert werden. Die Arbeit mit Abfrageausdrücken ist einfach, da sie viele vertraute Konstrukte der Sprache C# verwenden.
- Alle Variablen in einem Abfrageausdruck sind stark typisiert, obwohl Sie den Typ in vielen Fällen nicht explizit angeben müssen, da der Compiler ihn ableiten kann. Weitere Informationen finden Sie unter Typbeziehungen in LINQ-Abfragevorgängen.
- Eine Abfrage wird erst ausgeführt, wenn Sie die Abfragevariable durchlaufen, z.B. in einer foreach-Anweisung. Weitere Informationen finden Sie unter Einführung in LINQ-Abfragen.
- Zur Kompilierzeit werden Abfrageausdrücke gemäß den in der C#-Spezifikation festgelegten Regeln in Methodenaufrufe des Standardabfrageoperators konvertiert. Jede Abfrage, die mithilfe der Abfragesyntax ausgedrückt werden kann, kann auch mithilfe der Methodensyntax ausgedrückt werden. In den meisten Fällen ist aber die Abfragesyntax präziser und besser lesbar.Weitere Informationen finden Sie unter Spezifikation für die Sprache C# und Übersicht über Standardabfrageoperatoren.
- Beim Schreiben von LINQ-Abfragen empfiehlt sich diese Faustregel: Verwenden Sie die Abfragesyntax, wann immer es möglich ist, und verwenden Sie die Methodensyntax nur, wenn es nötig ist. Zwischen den beiden Formen gibt es keine semantischen oder leistungsbezogenen Unterschiede. Abfrageausdrücke sind oft besser lesbar als die entsprechenden in der Methodensyntax geschriebenen Ausdrücke.
- Für einige Abfragevorgänge, wie z.B. Count oder Max, gibt es keine entsprechende Abfrageausdrucksklausel, daher müssen diese als Methodenaufruf ausgedrückt werden. Die

Methodensyntax kann auf verschiedene Weise mit der Abfragesyntax kombiniert werden. Weitere Informationen finden Sie unter Abfragesyntax und Methodensyntax in LINQ.

- Abfrageausdrücke können in Ausdrucksbaumstrukturen oder Delegaten kompiliert werden, je nachdem, auf welchen Typ die Abfrage angewendet wird. IEnumerable<T>-Abfragen werden zu Delegaten kompiliert. IQueryable- und IQueryable<T>-Abfragen werden zu Ausdrucksbaumstrukturen kompiliert. Weitere Informationen finden Sie unter Ausdrucksbaumstrukturen.



Beispiele

A. Ohne LINQ

```
int[] numbers = {3,6,7,9,2,5,3,7};
int i = 0;

// Display numbers larger than 5
while (i < numbers.Length) {
  if(numbers[i]>5)
     Console.WriteLine(numbers[i]);
  ++i;
}
```

```
int[] numbers = {3,6,7,9,2,5,3,7};

var res = from n numbers
          where n > 5
          select n;

foreach (int n in res)
         Console.WriteLine(n);
```

B. Abfrageausdrücke

```
int[] numbers = { 7, 53, 45, 99 };
var res = from n in numbers
```

```
            where n > 50
            orderby n
            select n.ToString();
```

C. Lambadaausdrücke

```
int[] numbers = { 7, 53, 45, 99 };
var res = numbers
          .Where(n => n > 50)
          .OrderBy(n => n)
          .Select(n => n.ToString());
```

D. Abfrageausführung

An welchem Punkt Abfrageausdrücke ausgeführt werden ist unterschiedlich. LINQ-Abfragen werden beim Durchlaufen der Abfrage-Variablen ausgeführt, nicht bei der Erstellung. Dies wird als verzögerte Ausführung (deferred execution) bezeichnet. Eine sofortige Ausführung der Abfrage (immediate execution) kann auch erzwungen werden. Dies ist für die Speicherung der Zwischenergebnissen sinnvoll.

```
int[] numbers = { 1, 2, 3, 4, 5 };
var result = numbers.Where(n => n >= 2 && n <= 4);
Console.WriteLine(result.Max()); // <- query executes at this point
// Output:
//4
```

Die verzögerte Ausführung ermöglicht die Kombination mehrerer Abfragen oder die Erweiterung einer bestehenden Abfrage durch das Hinzufügen neuer Operationen. Die Änderungen werden dann bei der Ausführung der Abfrage berücksichtigt.

```
string[] words = { "one", "two", "three" };
var result = words
             .Select((w, i) => new { Index = i, Value = w })
             .Where(w => w.Value.Length == 3)
             .ToList();

Console.WriteLine("Prints index for words that have a string length of 3:");

foreach(var word in result)
       Console.WriteLine (word.Index.ToString());
// Output:
// Prints index for words that have a string length of 3:
// 0
// 1
```

Mit Objektinitialisierern kann man allen verfügbaren Feldern oder Eigenschaften eines Objekts zum Erstellungszeitpunkt Werte zuweisen, ohne einen Konstruktor gefolgt von Zeilen mit Zuweisungsanweisungen aufrufen zu müssen.

```
public class Student
{
```

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

    public Student() { }

    public Student(string name)
    {
        this.Name = name;
    }
}

Student bob = new Student { Age = 20, Name = "Bob" };
Student dob = new Student("Dob"){ Age = 19 };
```

Obwohl Objektinitialisierer in jedem Kontext verwendet werden können, sind sie vor allem in LINQ-Abfrageausdrücken nützlich. Abfrageausdrücke verwenden häufig anonyme Typen, die nur mit einem Objektinitialisierer initialisiert werden können. Anonyme Typen ermöglichen der select-Klausel in einem LINQ-Abfrageausdruck, Objekte der ursprünglichen Sequenz in Objekte zu transformieren, deren Wert und Form sich vom Original unterscheiden können. Dies ist nützlich, wenn man nur einen Teil der Informationen aus jedem Objekt in einer Sequenz speichern möchte.

```
var student = new { Age = 10, Name = "Bob" };
Console.WriteLine(student.GetType()); //<>f__AnonymousType1`2[System.Int32,System.String]
Console.WriteLine(student.Age + student.Name); // 10Bob
Console.WriteLine(student);  //{ Age = 10, Name = Bob }

List<Student> students = new List<Student> {
    new Student{ Name = "Bob", Age=20},
    new Student{ Name = "Dob", Age=19},
    new Student{ Name = "Lob", Age=20}
};

     var student_info = from s in students
                        select new { s.Name, s.Age };

     foreach(var s in student_info)
        Console.WriteLine (s.ToString());
```

Jedes Objekt im neuen anonymen Typ weist zwei öffentliche Eigenschaften auf, die die gleichen Namen wie die Eigenschaften oder Felder im ursprünglichen Objekt erhalten. Man kann ein Feld auch umbenennen, wenn man einen anonymen Typ erstellt.

```
select new {s.Name, StudentAge = s.Age};
```

Die verzögerte Ausführung erlaubt, Abfragen zu kombinieren oder zu erweitern. Im folgenden Beispiel wurde eine Basisabfrage erstellt und dann auf zwei neue separate Abfragen erweitert:

```
int[] numbers = { 1, 5, 10, 18, 23};
```

```
var baseQuery = from n in numbers select n;
//IEnumerable<int> baseQuery = from n in numbers select n*2;
var oddQuery = from b in baseQuery where b % 2 == 1 select b;

Console.WriteLine("Sum of odd numbers: " + oddQuery.Sum()); // <- query executes at this
point

var evenQuery = from b in baseQuery where b % 2 == 0 select b;
Console.WriteLine("Sum of even numbers: " + evenQuery.Sum()); // <- query executes at
this point

// Output:
// Sum of odd numbers: 29
// Sum of even numbers: 28
```

Operatoren

E. **Aggregate:** Wendet eine bestimmte Operation auf jedes Element in einer Sammlung an und hält das Ergebnis weiter

```
var numbers = new int[] { 1, 2, 3, 4, 5 };
var result = numbers.Aggregate((a, b) => a * b);

Console.WriteLine("Aggregated numbers by multiplication:");
Console.WriteLine(result);

//Output:
//Aggregated numbers by multiplication: //120
```

F. **Any:** Prüft, ob Elemente in einer Collection eine bestimmte Bedingung erfüllen

```
string[] names = { "Bob", "Ned", "Amy", "Bill" };
var result = names.Any(n => n.StartsWith("B"));

Console.WriteLine("Does any of the names start with the letter 'B':");
Console.WriteLine(result);
```

G. **ElementAtOrDefault:** Gibt ein Element am angegebenen Index zurück. Gibt den Standardwert zurück, wenn der Index außerhalb des Ranges liegt

```
string[] colors = { "Red", "Green", "Blue" };

var resultIndex1 = colors.ElementAtOrDefault(1);
var resultIndex10 = colors.ElementAtOrDefault(10);

Console.WriteLine("Element at index 1 in the array contains:"); Console.WriteLine(resultIndex1);

Console.WriteLine("Element at index 10 in the array does not exist:");
Console.WriteLine(resultIndex10 == null);
```

```
Output:
//Element at index 1 in the array contains: //Green
//Element at index 10 in the array does not exist: //True
```

H. **SelectMany:** Reduziert Collections zu einer einzigen Collection (ähnlich wie Cross-Join in SQL)

```
string[] fruits = { "Grape", "Orange", "Apple" };
int[] amounts = { 1, 2, 3 };

var result = fruits.SelectMany(f => amounts, (f, a) => new { Fruit = f,Amount = a });

Console.WriteLine("Selecting all values from each array, and mixing them:");
foreach (var o in result)
      Console.WriteLine(o.Fruit + ", " + o.Amount); }

Output:
Selecting all values from each array, and mixing them: Grape, 1
Grape, 2
Grape, 3
Orange, 1
Orange, 2
Orange, 3
Apple, 1
Apple, 2
Apple, 3
```

I. **ToDictionary:** Konvertiert eine Collection in ein Dictionary mit Schlüssel und Wert

```
class English2German {
      public string EnglishSalute { get; set; }
      public string GermanSalute { get; set; }
}

static void Sample_ToDictionary_Lambda_Simple() {
  English2German[] english2German = {
      new English2German { EnglishSalute = "Good morning", GermanSalute = "Guten Morgen" },
      new English2German { EnglishSalute = "Good day", GermanSalute = "Guten Tag" },
      new English2German { EnglishSalute = "Good evening", GermanSalute = "Guten Abend" },
  };

  var result = english2German.ToDictionary(k => k.EnglishSalute, v => v.GermanSalute);

  Console.WriteLine("Values inserted into dictionary:");
  foreach (KeyValuePair<string, string> dic in result)
      Console.WriteLine(String.Format("English salute {0} is {1} in German", dic.Key,
dic.Value)); }

Output:
Values put into dictionary:
English salute Good morning is Guten Morgen in German
```

```
English salute Good day is Guten Tag in German
English salute Good evening is Guten Abend in German
```

J. **AsEnumerable:** wandelt eine Collection in IEnumerable um

```
string[] names = { "John", "Suzy", "Dave" };
var result = names.AsEnumerable();

Console.WriteLine("Iterating over IEnumerable collection:");
foreach (var name in result)
	Console.WriteLine(name);
```

K. **Concat:** Verkettet zwei Collections

```
int[] numbers1 = { 1, 2, 3 };
int[] numbers2 = { 4, 5, 6 };

var result = numbers1.Concat(numbers2);

Console.WriteLine("Concatenating numbers1 and numbers2 gives:");
foreach (int number in result)
	Console.WriteLine(number);

Output:
Concatenating numbers1 and numbers2 gives: 1
2
3
4
5
6
```

L. **Distinct:** Entfernt doppelte Elemente

```
int[] numbers = { 1, 2, 2, 3, 5, 6, 6, 6, 8, 9 };
var result = (from n in numbers.Distinct() select n);

Console.WriteLine("Distinct removes duplicate elements:");

foreach (int number in result)
	Console.WriteLine(number);

Output:
Distinct removes duplicate elements: 1
2
3
5
6
8
9
```

M. **Except:** Entfernt alle Elemente, die in einer anderen Collection vorkommen

```
int[] numbers1 = { 1, 2, 3 };
int[] numbers2 = { 3, 4, 5 };

var result = (from n in numbers1.Except(numbers2) select n);

Console.WriteLine("Except creates a single sequence from numbers1 and removes the duplicates
found in numbers2:");
foreach (int number in result)
	Console.WriteLine(number);

Output:
Except creates a single sequence from numbers1 and removes the duplicates found in numbers2:
1
2
```

N. **GroupBy:** Gruppiert Elemente einer Collection nach Schlüssel

```
int[] numbers = { 10, 15, 20, 25, 30, 35 };
var result = from n in numbers
             group n by (n % 10 == 0) into groups
             select groups;

Console.WriteLine("GroupBy has created two groups:");

foreach (IGrouping<bool, int> group in result) {
	if (group.Key == true)
		Console.WriteLine("Divisible by 10");
	else
		Console.WriteLine("Not Divisible by 10");
	foreach (int number in group)
		Console.WriteLine(number);
}

Output:
GroupBy has created two groups:
Divisible by 10
10
20
30
Not Divisible by 10
15
25
35
```

O. **Intersect:** Nimmt nur die Elemente, die in beiden Collections vorkommen

```
int[] numbers1 = { 1, 2, 3 };
int[] numbers2 = { 3, 4, 5 };

var result = (from n in numbers1.Intersect(numbers2) select n);
```

```
Console.WriteLine("Intersect creates a single sequence with only the duplicates:");

foreach (int number in result)
      Console.WriteLine(number);

Output:
Intersect creates a single sequence with only the duplicates:
3
```

P. **Join:** Verbindet zwei Collections mit einem gemeinsamen Schlüsselwert (wie Inner Join)

```
string[] warmCountries = { "Turkey", "Italy", "Spain", "Saudi Arabia", "Etiopia" };
string[] europeanCountries = { "Denmark", "Germany", "Italy", "Portugal", "Spain" };

var result = (from w in warmCountries join e in europeanCountries on w equals e select w);

Console.WriteLine("Joined countries which are both warm and European using Query Syntax:");

foreach (var country in result)
      Console.WriteLine(country);

Output:
Joined countries which are both warm and European using Query Syntax:
Italy
Spain
```

Q. **OrderBy:** Sortiert aufsteigend

```
class Car{
      public string Name { get; set; }
      public int HorsePower { get; set; }
}

static void Sample_OrderBy_Linq_Objects() {
  Car[] cars = {
      new Car { Name = "Super Car", HorsePower = 215 },
      new Car { Name = "Economy Car", HorsePower = 75 },
      new Car { Name = "Family Car", HorsePower = 145 },
  };

  var result = from c in cars orderby c.HorsePower
               select c;

  Console.WriteLine("Ordered list of cars by horsepower using Query Syntax:");

  foreach (Car car in result)
      Console.WriteLine(String.Format("{0}: {1} horses", car.Name, car.HorsePower));
}

Output:
```

```
Ordered list of cars by horsepower using Query Syntax:
Economy Car: 75 horses
Family Car: 145 horses
Super Car: 215 horses
```

R. **Range:** Erzeugt eine Folge von numerischen Werten

```
var result = from n in Enumerable.Range(0, 10)
             select n;

Console.WriteLine("Counting from 0 to 9:");
foreach (int number in result)
       Console.WriteLine(number);

Output:
Counting from 0 to 9:
0
1
2
3
4
5
6
7
8
9
```

S. **Repeat:** Erstellt eine Sammlung wiederholter Elemente. Das erste Argument ist der zu wiederholende Wert und das zweite Argument ist die Anzahl der Wiederholungen.

```
string word = "Banana";
var result = from w in Enumerable.Repeat(word, 5) select w;

Console.WriteLine("String is repeated 5 times:");

foreach (string str in result)
       Console.WriteLine(str);

Output:
String is repeated 5 times:
Banana
Banana
Banana
Banana
Banana
```

T. **Reverse:** Kehrt die Elemente einer Collection um

```
char[] characters = { 's', 'a', 'm', 'p', 'l', 'e' };
var result = (from c in characters.Reverse() select c);
```

```
Console.WriteLine("Characters in reverse order:");
foreach (char character in result)
	Console.WriteLine(character);

Output:
Characters in reverse order:
e
l
p
m
a
s
```

U. **SkipWhile:** Überspringt Elemente, solange eine angegebene Bedingung erfüllt ist.

```
string[] words = { "one", "two", "three", "four", "five", "six" };
var result = words.SkipWhile(w => w.Length == 3);

Console.WriteLine("Skips words while the condition is met:");
foreach (string word in result)
	Console.WriteLine(word);

Output:
Skips words while the condition is met:
three
four
five
```

V. **Take:** Nimmt die angegebene Anzahl von Elementen, beginnend mit dem ersten Element.

```
int[] numbers = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };
var result = numbers.Take(5);

Console.WriteLine("Takes the first 5 numbers only:");
foreach (int number in result)
	Console.WriteLine(number);

Output:
Takes the first 5 numbers only:
1
2
3 4 5
```

W. **ThenBy:** Sortiert eine bereits sortierte Sammlung nach weiteren Kriterien

```
string[] capitals = { "Berlin", "Paris", "Madrid", "Tokyo", "London", "Athens", "Beijing",
"Seoul" };

var result = (from c in capitals orderby c.Length select c) .ThenBy(c => c);

Console.WriteLine("Ordered list of capitals, first by length and then alphabetical:");
```

```
foreach (string capital in result)
      Console.WriteLine(capital);

Output:
Ordered list of capitals, first by length and then alphabetical:
Paris
Seoul
Tokyo
Athens
Berlin
London
Madrid
Beijing
```

X. **Union:** Kombiniert zwei Collections und entfernt alle Wiederholungen

```
int[] numbers1 = { 1, 2, 3 };
int[] numbers2 = { 3, 4, 5 };
var result = (from n in numbers1.Union(numbers2) select n);

Console.WriteLine("Union creates a single sequence and eliminates the duplicates:");
foreach (int number in result)
      Console.WriteLine(number);

Output:
Union creates a single sequence and eliminates the duplicates:
1
2
3
4
5
```

Y. **Zip:** Wendet eine Funktion parallel auf Elemente aus zwei Collections an und kombiniert das Ergebnis

```
int[] numbers1 = { 1, 2, 3 };
int[] numbers2 = { 10, 11, 12 };

var result = numbers1.Zip(numbers2, (a, b) => (a * b));

Console.WriteLine("Using Zip to combine two arrays into one (1*10, 2*11, 3*12):");
foreach (int number in result)
      Console.WriteLine(number);

Output:
Using Zip to combine two arrays into one (1*10, 2*11, 3*12):
10
22
36
```

Z. **XML**

```
string myXML = @"<Company>
                    <Departments>
                        <Department name=""Accounting""><Lead>BOB</Lead></Department>
                        <Department name=""Sales""><Lead>DOB</Lead></Department>
                        <Department name=""Marketing""><Lead>LOB</Lead></Department>
                    </Departments>
                </Company>";

XElement xdoc = XElement.Parse(myXML);

var names =  from dept in xdoc.Descendants("Department")
                                select (string) dept.Attribute("Name");

names.ToList().ForEach(Console.WriteLine);

Output:
Accounting
Sales
Marketing
```

2. Übungen

1. Gegeben sei eine Liste von Wörter. Schreiben Sie eine Abfrage, die alle Wörter mit mindestens 4 Zeichen als Großbuchstaben zurückgibt.

   Input: this, is, a, not, very, long, test

   Output: THIS, VERY, LONG, TEST

2. Gegeben sei ein Array mit Zahlen. Geben Sie alle Zahlen des Arrays zusammen mit der Anzahl von Erscheinungen aus

   Input: 2 3 4 5 4 23 5 4 4 2 4 3 5

   Output: 2 2, 3 2, 4 5, 5 3, 23 1

3. Geben Sie die großgeschriebenen Wörter eines String aus

   Input: “This IS a TEST String”

   Output: IS, TEST

4. Berechnen Sie das Skalarprodukt zwei Arrays

   Input: [1,2,3], [4,5,6]

   Output: 32

5. Gegeben sei eine Liste von Wörter. Schreiben Sie eine Abfrage, die alle Wörter zurückgibt, die mindestens 4 Zeichen haben und mit den angegebenen Buchstaben beginnen und enden.

   Input: this, is, a, not, very, long, test

   l, g

   Output: long

6. Gegeben sei das folgende XML-Dokument

```
<?xml version="1.0"?>
<Shop>
  <Products>
    <Product id="11">
      <Name>Laptop</Name>
      <Quantity>1</Quantity>
      <Price>5000</Price>
    </Product>
    <Product id="13">
      <Name>Keyboard</Name>
      <Quantity>12</Quantity>
      <Price>1000</Price>
    </Product>
  </Products>
</Shop>
```

   Geben Sie eine nach id sortierte Liste von Products aus, deren Gesamtpreis (berechnet als Quantity*Price) größer als 6000 ist.