

Lambda Expression, LINQ API



Lesson Objectives

- Understand Lambda Expression
- Linq API

Section 1

LAMBDA EXPRESSIONS

- A *lambda expression* is an expression of any of the following two forms:
 - ✓ Expression lambda that has an expression as its body:

(input-parameters) => expression

- ✓ Statement lambda that has a statement block as its body:

(input-parameters) => { <sequence-of-statements> }

Expression Lambdas 1-3

- An expression lambda is a lambda with an expression on the right side.

Syntax

```
(input_parameters) => expression
```

where,

input_parameters: one or more input parameters, each separated by a comma

expression: the expression to be evaluated

- The input parameters may be implicitly typed or explicitly typed.
- When there are two or more input parameters on the left side of the lambda operator, they must be enclosed within parentheses. However, if you have only one input parameter and its type is implicitly known, then the parentheses can be omitted. For example,

```
(str, str1) => str == str1
```

- ◆ It means **str** and **str1** go into the comparison expression which compares **str** with **str1**. In simple terms, it means that the parameters **str** and **str1** will be passed to the expression **str == str1**.
- ◆ Here, it is not clear what are the types of **str** and **str1**.
- ◆ Hence, it is best to explicitly mention their data types:

```
(string str, string str1)=> str==str1
```

Expression Lambdas 2-3

- To use a lambda expression:
 - ✓ Declare a delegate type which is compatible with the lambda expression.
 - ✓ Then, create an instance of the delegate and assign the lambda expression to it. After this, you will invoke the delegate instance with parameters, if any.
 - ✓ This will result in the lambda expression being executed. The value of the expression will be the result returned by the lambda.
- The following code demonstrates expression lambdas:

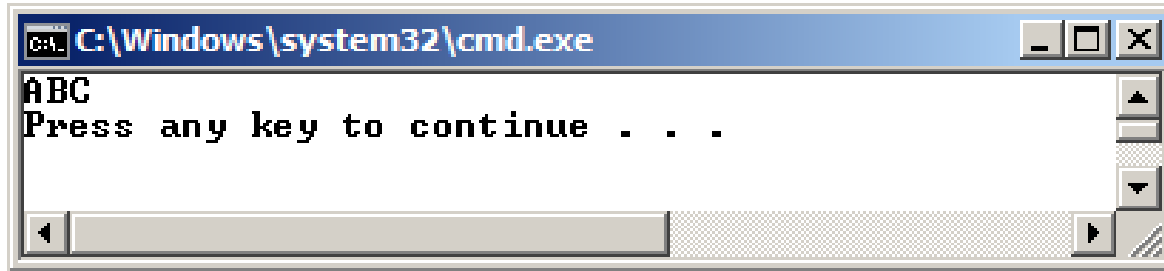
Snippet

```
/// <summary>  
/// Class ConvertString converts a given string to uppercase  
/// </summary>  
public class ConvertString{  
    delegate string MakeUpper(string s);  
    public static void Main() {  
        // Assign a lambda expression to the delegate instance  
        MakeUpper con = word => word.ToUpper();  
        // Invoke the delegate in Console.WriteLine with a string  
        // parameter  
        Console.WriteLine(con("abc"));  
    }  
}
```

Expression Lambdas 3-3

- In the code:
 - ✓ A delegate named **MakeUpper** is created and instantiated. At the time of instantiation, a lambda expression, `word => word.ToUpper()` is assigned to the delegate instance.
 - ✓ The meaning of this lambda expression is that, given an input, **word**, call the `ToUpper()` method on it.
 - ✓ `ToUpper()` is a built-in method of `String` class and converts a given string into uppercase.
- The following figure displays the output of using expression lambdas:

Output



- A statement lambda is a lambda with one or more statements. It can include loops, if statements, and so forth.

Syntax

```
(input_parameters) => {statement;;}
```

- where,

- ✓ **input_parameters**: one or more input parameters, each separated by a comma
- ✓ **statement**: a statement body containing one or more statements
- ✓ Optionally, you can specify a return statement to get the result of a lambda.
- ✓ `(string str, string str1)=> { return (str==str1); }`

- The following code demonstrates a statement lambda expression:

Snippet

```
/// <summary>  
/// Class WordLength determines the length of a given word or phrase  
/// </summary>  
public class WordLength{  
    // Declare a delegate that has no return value but accepts a string  
    delegate void GetLength(string s);  
    public static void Main() {  
        // Here, the body of the lambda comprises two entire statements  
        GetLength len = name => { int n =  
            name.Length; Console.WriteLine(n.ToString()); };  
        // Invoke the delegate with a string  
        len("Mississippi");  
    }  
}
```


Lambdas with Standard Query Operators 1-2

- Lambda expressions can also be used with standard query operators.
- The following table lists the standard query operators:

Operator	Description
Sum	Calculates sum of the elements in the expression
Count	Counts the number of elements in the expression
OrderBy	Sorts the elements in the expression
Contains	Determines if a given value is present in the expression

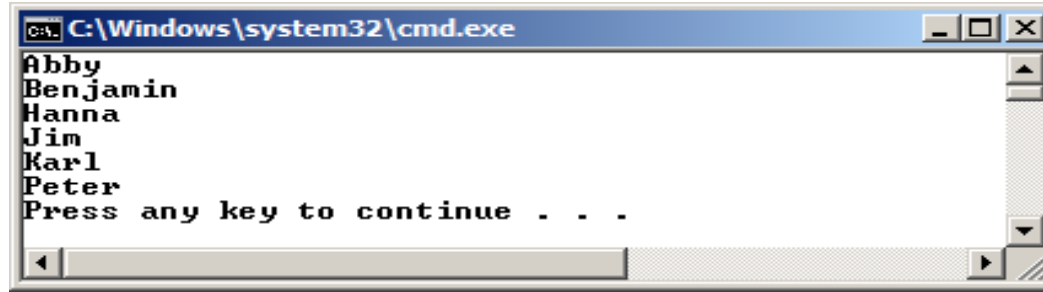
- The following shows how to use the `OrderBy` operator with the lambda operator to sort a list of names:

Snippet

```
/// <summary>
/// Class NameSort sorts a list of names
/// </summary>
public class NameSort{
    public static void Main() {
        // Declare and initialize an array of strings
        string [ ] names = {"Hanna", "Jim", "Peter", "Karl", "Abby",
            "Benjamin"};
        foreach (string n in names.OrderBy(name => name)) {
            Console.WriteLine(n);
        }
    }
}
```

Lambdas with Standard Query Operators 2-2

- The following figure displays the output of the `OrderBy` operator example:



A screenshot of a Windows command prompt window titled "C:\Windows\system32\cmd.exe". The window displays the following text: Abby, Benjamin, Hanna, Jim, Karl, Peter, and Press any key to continue . . . The text is displayed in a monospaced font. The window has a standard Windows XP-style title bar with minimize, maximize, and close buttons.

Lambda Expression - Summary

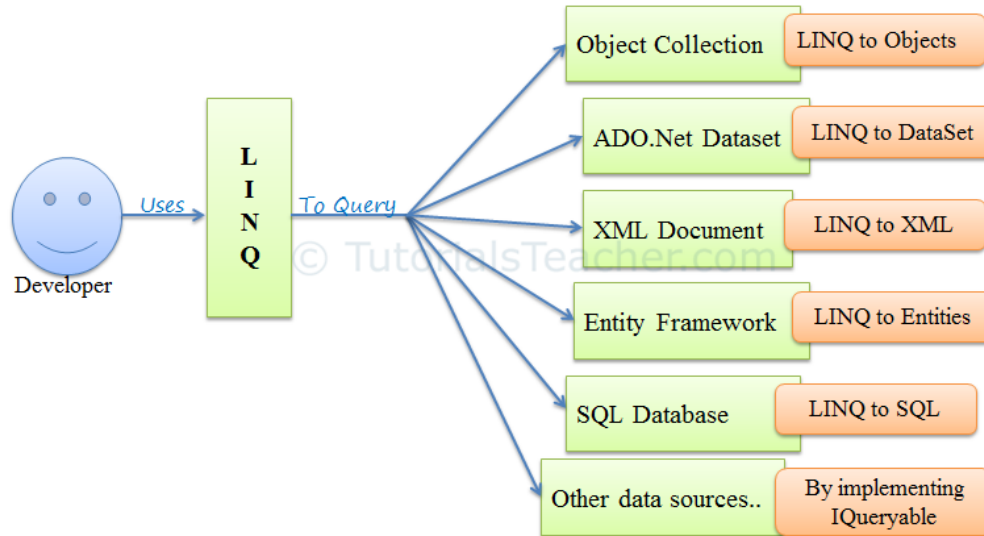
- Use the lambda declaration operator `=>` to separate the lambda's parameter list from its body. To create a lambda expression, you specify input parameters (if any) on the left side of the lambda operator and an expression or a statement block on the other side.

Section 2

LANGUAGE INTEGRATED QUERY (LINQ)

What's LINQ?

- Language Integrated Query
- What's LINQ? Language Integrated Query (LINQ) is a query syntax built in C# and VB.NET used to save and retrieve data from different types of data sources like an ObjectCollection, SQL server database, XML, web service etc.



Advantages

- Syntax highlighting
- Easy debugging
- Extensible that means it is possible to query new data sourcetypes.
- Facility of joining several data sources in a single query.
- Easy transformation (like transforming SQL data to XML data .)

LINQ operators

- Filtering Operators
- Join Operators
- Projection Operations
- Sorting Operators
- Grouping Operators
- Conversions
- Concatenation
- Aggregation
- Quantifier Operations
- Partition Operations
- Generation Operations
- Set Operations
- Equality
- Element Operators

Query Syntax

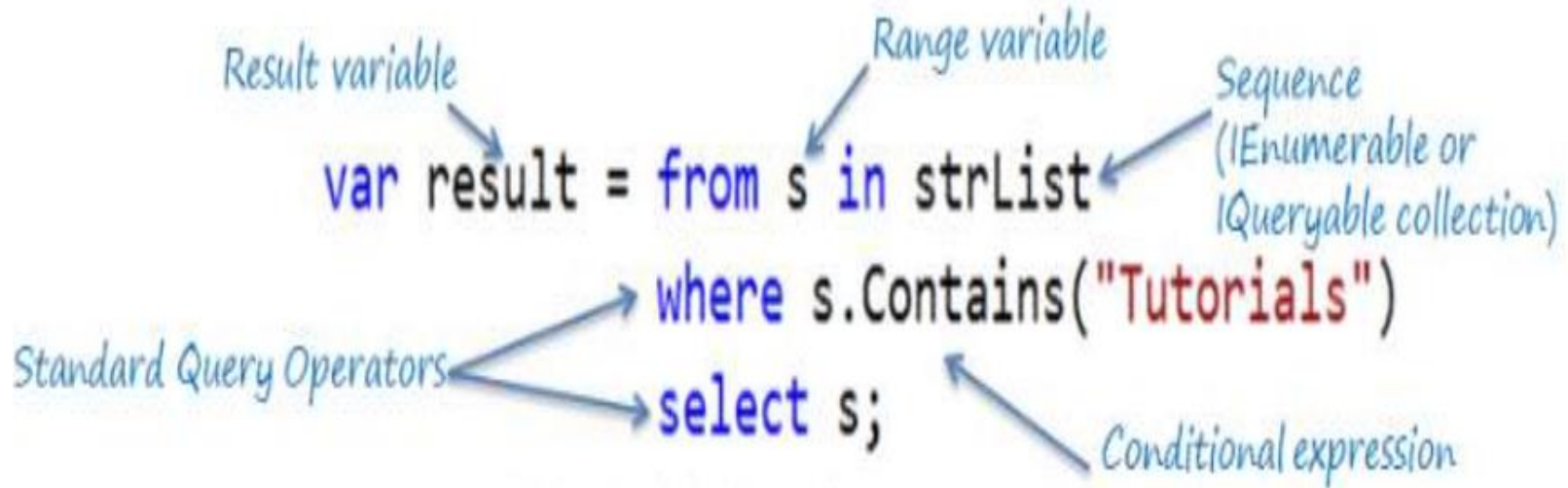
- Query syntax is similar to SQL (Structured Query Language) for the database.
- It is defined within the C #.
- Query syntax starts with from clause and can be end with Select or GroupBy clause.
- Implicitly typed variable - var can be used to hold the result of the LINQ query.
- We use various operators like filtering, joining, grouping, sorting operators to construct the desired result.

Example: LINQ Query syntax in C#

```
//string collection
IList<string> stringList = new List<string> {
    "C# Tutorials",
    "VB.NET Tutorials",
    "Learn C++",
    "MVC Tutorials",
    "Java Introduction"
};

//LINQ Query Syntax
var result = from s in stringList
              where s.Contains("Tutorials")
              select s;
```

LINQ Query Syntax



The diagram illustrates the LINQ query syntax with the following code and annotations:

```
var result = from s in strList
              where s.Contains("Tutorials")
              select s;
```

Annotations:

- Result variable:** Points to `var result`.
- Range variable:** Points to `s` in `from s in strList`.
- Sequence (IEnumerable or IQueryable collection):** Points to `strList`.
- Standard Query Operators:** Points to `where` and `select`.
- Conditional expression:** Points to `s.Contains("Tutorials")`.

© TutorialsTeacher.com

Method Syntax

- **Method Syntax** is like calling extension method already included
- Implicitly typed variable - var can be used to hold the result of the LINQ query.

```
//string collection
IList<string> stringList = new List<string> {
    "C# Tutorials",
    "VB.NET Tutorials",
    "Learn C++",
    "MVC Tutorials",
    "Java Introduction"
};

var result = stringList.Where(s => s.Contains("Tutorials"));
```

TutorialsTeacher.com

```
var result = strList.Where(s => s.Contains("Tutorials"));
```

Extension method

Lambda expression

Filtering Operators

- Filtering is an operation to restrict the result set such that it has only selected elements satisfying a Where particular condition

Operator	Description	C# Query Expression Syntax
where	Filter values based on a predicate function	where

```
string[] words = { "humpty", "dumpty", "set", "on", "a", "wall"};
IEnumerable<string> query = from word in words
                           where word.Length == 3
                           select word;

foreach (var str in query)
{
    Console.WriteLine(str);
}
```

Grouping Operations

- The operators put data into some groups based on a common GroupBy shared attribute

Operator	Description	C# Query Expression Syntax
GroupBy	Organize a sequence of items in groups and return them as an IEnumerable collection of type IGrouping<key, element>	group ... by -or- group ... by ... into ...

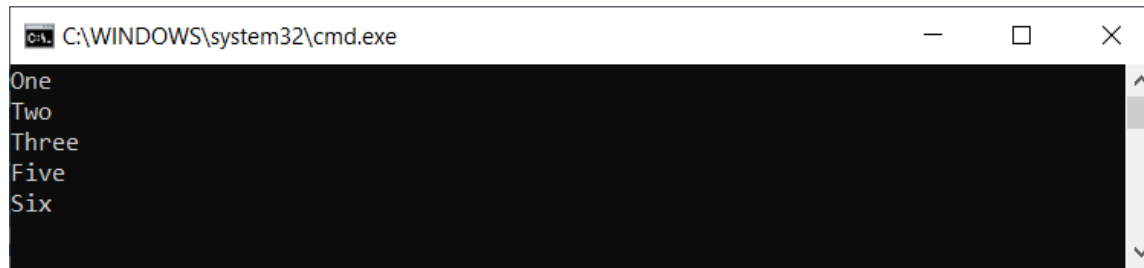
```
List<int> numbers = new List<int>() {35,44,200,84,3987,4,199,329,446,208 };
IEnumerable<IGrouping<int, int>> query = from number in numbers
                                         group number by number % 2;

foreach (var group in query)
{
    Console.WriteLine(group.Key==0?"\nEvent numbers:":"\nOdd numbers:");
    foreach (var number in group)
    {
        Console.Write(number + ", ");
    }
}
```

Concatenation Operations

- Performs concatenation of two sequences and is quite similar to the Union operator in terms of its operation except of the fact that this does not remove duplicates.

```
IList<string> collection1 = new List<string>() { "One", "Two", "Three" };  
IList<string> collection2 = new List<string>() { "Five", "Six" };  
  
var collection3 = collection1.Concat(collection2);  
  
foreach (string str in collection3)  
{  
    Console.WriteLine(str);  
}
```



C:\WINDOWS\system32\cmd.exe

```
One  
Two  
Three  
Five  
Six
```

Sorting Operators

- A sorting operation allows ordering the elements of a sequence on basis of a single or OrderBy, OrderByDescending, more attributes.

Sorting Operator	Description
OrderBy	Sorts the elements in the collection based on specified fields in ascending or descending order.
OrderByDescending	Sorts the collection based on specified fields in descending order. Only valid in method syntax.
ThenBy	Only valid in method syntax. Used for second level sorting in ascending order.
ThenByDescending	Only valid in method syntax. Used for second level sorting in descending order.
Reverse	Only valid in method syntax. Sorts the collection in reverse order.

Sorting Operators

```

IList<Student> studentList = new List<Student>() {
    new Student() { StudentId = 1, StudentName = "John", Age = 18 } ,
    new Student() { StudentId = 2, StudentName = "Steve", Age = 15 },
    new Student() { StudentId = 3, StudentName = "Bill", Age = 25 } ,
    new Student() { StudentId = 4, StudentName = "Ram" , Age = 20 } ,
    new Student() { StudentId = 5, StudentName = "Ron" , Age = 19 }
};

var orderByResult = from s in studentList
                    orderby s.StudentName
                    select s;

var orderByDescendingResult = from s in studentList
                               orderby s.StudentName descending
                               select s;

```


- Compares two sentences (enumerable) and determine are SequenceEqual they an exact match or not.

```
Student john = new Student() { StudentId=1,StudentName="John",Age=19};  
Student steve = new Student() { StudentId = 2, StudentName = "Steve", Age = 20 };  
Student bill = new Student() { StudentId = 3, StudentName = "Bill", Age = 21 };  
  
List<Student> students1 = new List<Student>() { john, steve };  
List<Student> students2 = new List<Student>() { john, steve };  
List<Student> students3 = new List<Student>() { john, steve,bill };  
  
bool equal = students1.SequenceEqual(students2);  
bool notEqual = students1.SequenceEqual(students3);
```

LINQ to Objects 1-3

- LINQ to Objects offers a fresh approach to collections
- Before we had to write long coding (foreach loops of much complexity) for retrieval of data from a collection
- Now we write declarative code which clearly describes the desired data that is required to retrieve.
- LINQ to Objects offers a fresh approach to collections Before we had to write long coding (foreach loops of much complexity) for retrieval of data from a collection Now we write declarative code which clearly describes the desired data that is required to retrieve.
 - ✓ LINQ to Objects offers a fresh approach to collections
 - ✓ Before we had to write long coding (foreach loops of much complexity) for retrieval of data from a collection
 - ✓ Now we write declarative code which clearly describes the desired data that is required to retrieve.

LINQ to Objects 2-3

```
1  using System;
2  using System.Collections.Generic;
3  using System.Linq;
4  using System.Text;
5
6  namespace LINQtoObjects
7  {
8      class Program
9      {
10         static void Main(string[] args)
11         {
12             string[] tools = { "Tablesaw", "Bandsaw", "Planer", "Jointer", "Drill",
13                               "Sander" };
14             var list = from t in tools
15                       select t;
16
17             StringBuilder sb = new StringBuilder();
18
19             foreach (string s in list)
20             {
21                 sb.Append(s + Environment.NewLine);
22             }
23             Console.WriteLine(sb.ToString(), "Tools");
24             Console.ReadLine();
25         }
26     }
27 }
```

LINQ to Objects 3-3

- In the example, an array of strings (tools) is used as the collection of objects

```
string[] tools = { "Tablesaw", "Bandsaw", "Planer", "Jointer", "Drill",  
                  "Sander" };
```

- Queried using LINQ to Objects

```
Objects query is:  
var list = from t in tools  
           select t;
```

- The output will be ..

Tablesaw

Bandsaw

Planer

Jointer

Drill

Sander

- (LINQ) is not only about retrieving data.
- We can use a source sequence as input and modify it in many ways to create a new output sequence.
- We Can ...
 - ✓ Merge multiple input sequences into a single output sequence that has a new type.
 - ✓ Create output sequences whose elements consist of only one or several properties of each element in the source sequence.
 - ✓ Create output sequences whose elements consist of the results of operations performed on the source data.
 - ✓ Create output sequences in a different format. For example, you can transform data from SQL rows or text files into XML.

Joining Multiple Inputs into One Output Sequence 1-3

```
class Student
{
    public string First { get; set; }
    public string Last { get; set; }
    public int ID { get; set; }
    public string Street { get; set; }
    public string City { get; set; }
    public List<int> Scores;
}
```

```
class Teacher
{
    public string First { get; set; }
    public string Last { get; set; }
    public int ID { get; set; }
    public string City { get; set; }
}
```

Joining Multiple Inputs into One Output Sequence 2-3

```
List<Student> students = new List<Student>()
{
    new Student {First="Svetlana",
        Last="Omelchenko",
        ID=111,
        Street="123 Main Street",
        City="Seattle",
        Scores= new List<int> {97, 92, 81, 60}},
    new Student {First="Claire",
        Last="O'Donnell",
        ID=112,
        Street="124 Main Street",
        City="Redmond",
        Scores= new List<int> {75, 84, 91, 39}},
    new Student {First="Sven",
        Last="Mortensen",
        ID=113,
        Street="125 Main Street",
        City="Lake City",
        Scores= new List<int> {88, 94, 65, 91}},
};
```

```
List<Teacher> teachers = new List<Teacher>()
{
    new Teacher {First="Ann", Last="Beebe", ID=945, City = "Seattle"},
    new Teacher {First="Alex", Last="Robinson", ID=956, City = "Redmond"},
    new Teacher {First="Michiyo", Last="Sato", ID=972, City = "Tacoma"}
};
```

Joining Multiple Inputs into One Output Sequence 3-3

```
var peopleInSeattle = (from student in students
                        where student.City == "Seattle"
                        select student.Last)
                        .Concat(from teacher in teachers
                                where teacher.City == "Seattle"
                                select teacher.Last);

Console.WriteLine("The following students and teachers live in Seattle:");
// Execute the query.
foreach (var person in peopleInSeattle)
{
    Console.WriteLine(person);
}

Console.WriteLine("Press any key to exit.");
Console.ReadKey();
```

- **Output will be...** The following students and teachers live in Seattle:
Omelchenko
Beebe

Create output sequences whose elements consist of only one or several properties of each element in the source sequence. 1-2

- Select one member of the source element

```
var query = from cust in Customers
             select cust.City;
```

- Create elements that contain more than one property from the source element

```
var query = from cust in Customer
             select new {Name = cust.Name, City = cust.City};
```

Create output sequences whose elements consist of only one or several properties of each element in the source sequence. 2-2

```
class FormatQuery
{
    static void Main()
    {
        // Data source.
        double[] radii = { 1, 2, 3 };

        // Query.
        IEnumerable<string> query =
            from rad in radii
            select String.Format("Area = {0}", (rad * rad) * 3.14);

        // Query execution.
        foreach (string s in query)
            Console.WriteLine(s);

        // Keep the console open in debug mode.
        Console.WriteLine("Press any key to exit.");
        Console.ReadKey();
    }
}
```

Create output sequences in a different format.

```
109     class XMLTransform
110     {
111     static void Main()
112     {
113         // Create the data source by using a collection initializer.
114         // The Student class was defined previously in this topic.
115         List<Student> students = new List<Student>()
116         {
117             new Student {First="Svetlana", Last="Omelchenko", ID=111, Scores = new List<int>{97, 92, 81, 60}},
118             new Student {First="Claire", Last="O'Donnell", ID=112, Scores = new List<int>{75, 84, 91, 39}},
119             new Student {First="Sven", Last="Mortensen", ID=113, Scores = new List<int>{88, 94, 65, 91}},
120         };
121
122         // Create the query.
123         var studentsToXML = new XElement("Root",
124             from student in students
125             let x = String.Format("{0},{1},{2},{3}", student.Scores[0],
126                 student.Scores[1], student.Scores[2], student.Scores[3])
127             select new XElement("student",
128                 new XElement("First", student.First),
129                 new XElement("Last", student.Last),
130                 new XElement("Scores", x)
131             ) // end "student"
132         ); // end "Root"
133
134         // Execute the query.
135         Console.WriteLine(studentsToXML);
136
137         // Keep the console open in debug mode.
138         Console.WriteLine("Press any key to exit.");
139         Console.ReadKey();
140     }
141 }
```

LINQ - Summary

- <Recap of the main points of Content #1>
- <Q&A>

References

- <http://www.tutorialspoint.com/linq/>
- <https://msdn.microsoft.com/en-us/library/mt693042.aspx>
- https://en.wikipedia.org/wiki/Language_Integrated_Query

Lesson Summary

- <Summarize the main points in the lesson, compared to the lesson objectives>

Thank you

