Module 10

"LINQ"

TEKNOLOGISK
INSTITUT

Agenda

- Extension Methods
- Anonymous Types
- Introducing LINQ
- First Look at LINQ Query Expressions
- LINQ Query Keywords
- More Query Operator Methods
- Query Variations

Defining Extension Methods

- *Extension methods* let you extend types with your own methods
  - Even if you don’t have the source or the types are not yours

```
static class MyExtensions
{
   public static string ToMyTimestamp( this DateTime dt )
   {
      return dt.ToString( "yyyy-MM-dd HH:mm:ss.fff" );
   }
}
```

- Must be **static** and defined in a **static** class
- The first parameter contains **this** and determines the type being extended

- Extension methods can have any number of parameters

Invoking Extension Methods

- Extension methods can be invoked at the instance level

```
DateTime dt = DateTime.Now;
Console.WriteLine( dt.ToMyTimestamp() );
```

- Alternatively, the method can be invoked statically

```
DateTime dt = DateTime.Now;
Console.WriteLine( MyExtensions.ToMyTimestamp( dt ) );
```

- Visual Studio 2012 has special IntelliSense for extension methods

Using Extension Methods

- The static class containing the extension methods must be in scope for the extension methods to be used

- Extension methods are indeed extending – not inheriting!
  - No access to private or protected members
  - All access is through the supplied parameter

```
public static string ToMyTimestamp( this DateTime dt )
{
   return dt.ToString( "yyyy-MM-dd HH:mm:ss.fff" );
}
```

- Can extend interfaces as well, but implementation must be provided

Agenda

TEKNOLOGISK
INSTITUT

- Extension Methods
- **Anonymous Types**
- Introducing LINQ
- First Look at LINQ Query Expressions
- LINQ Query Keywords
- More Query Operator Methods
- Query Variations

Creating Anonymous Types

- Combining implicitly typed variables with object initializer syntax provides an excellent shorthand for defining simple classes called *anonymous types*

```
var myEquipment = new { Manufacturer = "Nintendo",
                        Make = "Wii",
                        Controllers = 4 };
Console.WriteLine( "I have a {0} {1} with {2} controllers",
   myEquipment.Manufacturer,
   myEquipment.Make,
   myEquipment.Controllers );
```

- The compiler autogenerates an anonymous class for us to use
- This class inherits from `System.Object`

- Members are read-only!

Equality of Anonymous Types

- Anonymous types come with their own overrides of `System.Object` methods
  - `ToString()`
  - `Equals()`
  - `GetHashCode()`

- The `==` and `!=` operators are however not overloaded with `Equals()`!
  - The exact references are still compared

Restrictions to Anonymous Types

- Anonymous types can be nested arbitrarily

```
var myFancyEquipment = new
{
   Manufacturer = "Microsoft",
   Make = "Xbox 360",
   XboxLive = new { Name = "Komatoze",
                    Membership = MembershipType.Gold }
};
```

- Some restrictions do apply to anonymous types
  - Type name is auto-generated and cannot be changed
  - Always derive directly from `System.Object`
  - Fields and properties of anonymous types are always read-only
  - Anonymous types are implicitly sealed
  - No possibility of Custom methods, operators, overrides, or events

Agenda

- Extension Methods
- Anonymous Types
- **Introducing LINQ**
- First Look at LINQ Query Expressions
- LINQ Query Keywords
- More Query Operator Methods
- Query Variations

Motivation for LINQ

- LINQ = Language INtegrated Query

- Several distinct motivations for LINQ
  - Uniform programming model for any kind of data
  - A better tool for embedding SQL queries into type-safe code
  - Another data abstraction layer
  - …

- All of these descriptions to some extent hold true

LINQ Components

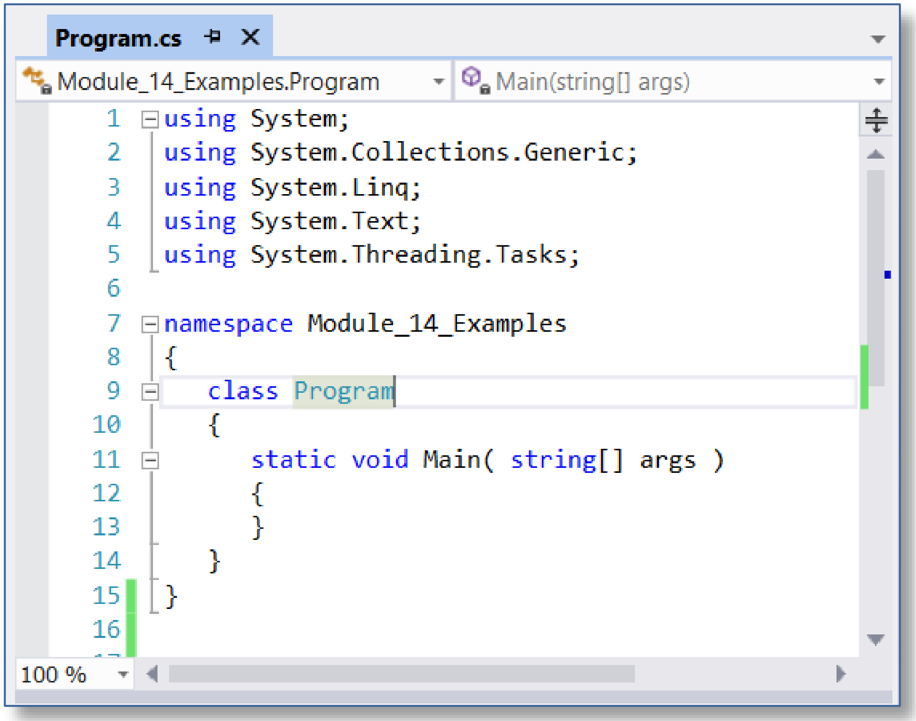
- LINQ to Objects
- LINQ to XML
- LINQ to SQL
- LINQ to DataSet
- LINQ to Entities
- Parallel LINQ
- …

- We will focus on LINQ to Objects in this module

- Later we will see
  - LINQ to XML
  - LINQ to Entities

Starting LINQ to Objects

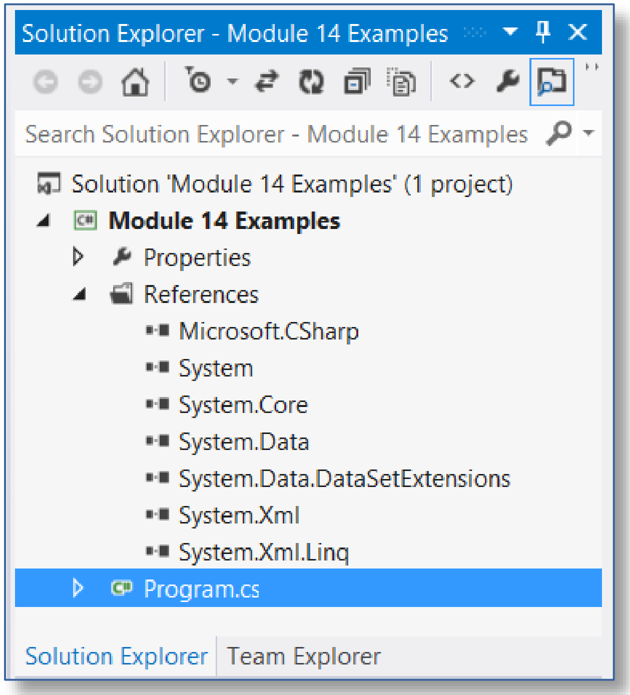
- Main LINQ features live in **`System.Core.dll`** in the **`System.Linq`** namespace



```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace Module_14_Examples
{
    class Program
    {
        static void Main( string[] args )
        {
        }
    }
}

```


Solution Explorer - Module 14 Examples
Search Solution Explorer - Module 14 Examples
Solution Explorer
Team Explorer

- Solution 'Module 14 Examples' (1 project)
  - **Module 14 Examples**
    - Properties
    - References
      - Microsoft.CSharp
      - System
      - System.Core
      - System.Data
      - System.Data.DataSetExtensions
      - System.Xml
      - System.Xml.Linq
    - Program.cs

Agenda

TEKNOLOGISK
INSTITUT

- Extension Methods
- Anonymous Types
- Introducing LINQ
- **First Look at LINQ Query Expressions**
- LINQ Query Keywords
- More Query Operator Methods
- Query Variations

A First Example

- Find all games with more that 18 characters in the title

```
string[] wiiGames = {
   "Super Mario Galaxy",
   "FIFA 09",
   "Guitar Hero III",
   "Wii Sports",
   "Wii Fit",
   "Legend of Zelda: Twilight Princess"
};

IEnumerable<string> query = from g in wiiGames
                            where g.Length >= 18
                            select g;
```

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

Implicitly Typed Variables

- Query results can be of a multitude of types

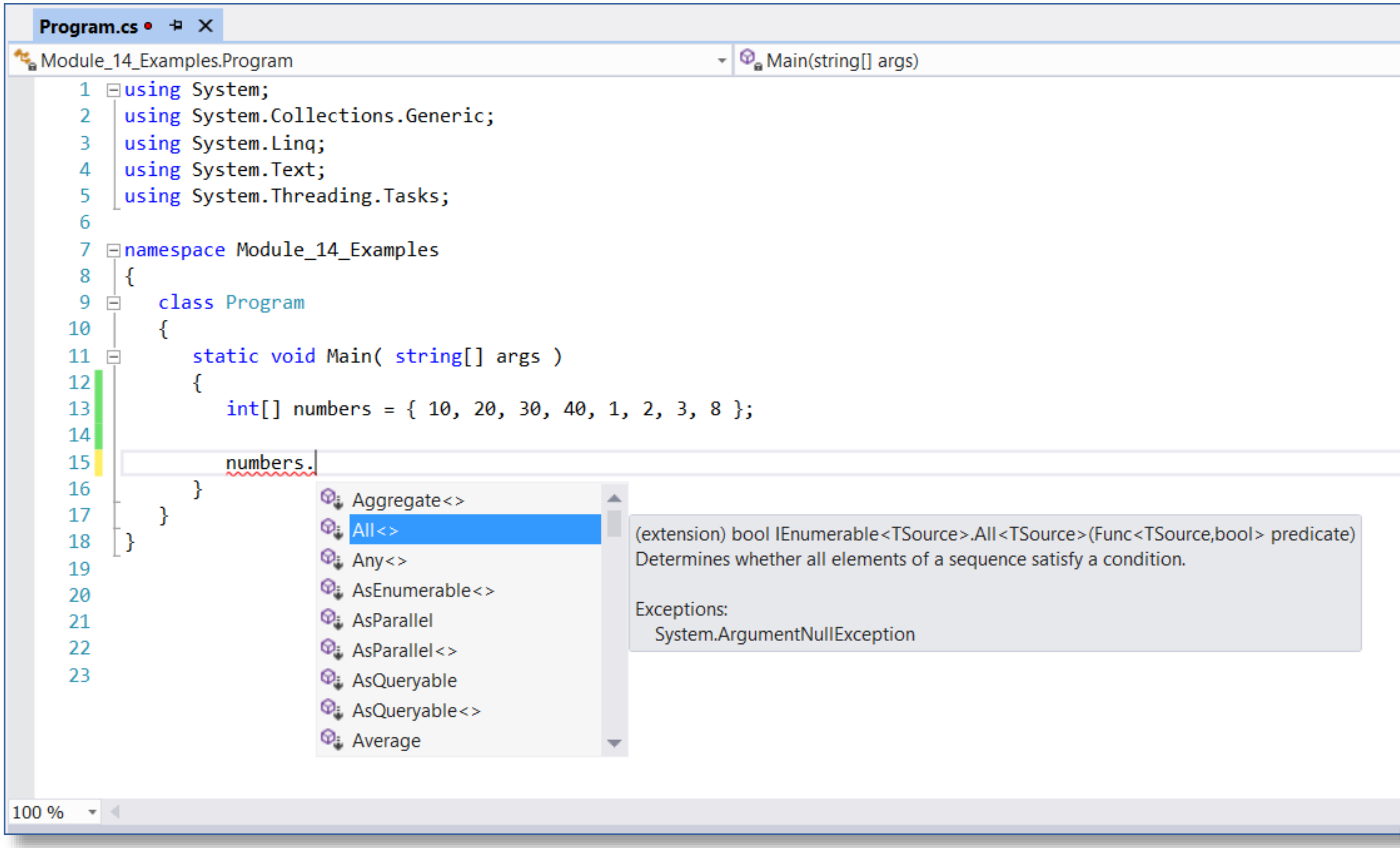
```
int[] numbers = {10, 20, 30, 40, 1, 2, 3, 8};
IEnumerable<int> query = from i in numbers
                         where i < 10 select i;
foreach( int i in query )
{
   Console.WriteLine( i );
}
```

- Innocently-looking modifications might change underlying type
- Make all query variables implicitly typed…!

```
int[] numbers = {10, 20, 30, 40, 1, 2, 3, 8};
var query = from i in numbers where i < 10 select i;
foreach( var i in query )
{
   Console.WriteLine( i );
}
```

Enumerable Extension Methods

- The **`System.Linq.Enumerable`** class provides a lot of extension methods



```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace Module_14_Examples
{
    class Program
    {
        static void Main( string[] args )
        {
            int[] numbers = { 10, 20, 30, 40, 1, 2, 3, 8 };

            numbers.
        }
    }
}
```

(extension) bool IEnumerable<TSource>.All<TSource>(Func<TSource,bool> predicate)
Determines whether all elements of a sequence satisfy a condition.

Exceptions:
System.ArgumentNullException

Deferred Execution

- Query expressions are not evaluated until they're enumerated!
- This is called *Deferred Execution*

```
int[] numbers = { 10, 20, 30, 40, 0, 1, 2, 3, 8 };
var query = from i in numbers where i < 10 select 87 / i;

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

- You can force evaluation through the Visual Studio 2012 debugger
  - Use the Results View of the query variable

Immediate Execution

- You can force evaluation by using conversion extension methods

```
int[] numbers = { 10, 20, 30, 40, 0, 1, 2, 3, 8 };
var query = from i in numbers where i < 10 select i;

int[] intNumbers = query.ToArray();
List<int> listNumbers = query.ToList();
```

- There are other such extension methods, e.g.
  - `ToDictionary<T,K>`

LINQ and Generic Collections

- LINQ can query data in various members of `System.Collections.Generic`

```
Stack<int> stack = new Stack<int>( new int[]{ 42, 87, 112, 255 } );
var query = from i in stack where i < 100 select i;
```

```
List<Car> cars = new List<Car>() {
   new Car{ PetName="Henry", Color="Silver", Speed=100, Make="VW" },
   new Car{ PetName="Daisy", Color="Tan", Speed=90, Make="BMW" },
   new Car{ PetName="Mary", Color="Black", Speed=55, Make="VW" },
   new Car{ PetName="Clunker", Color="Rust", Speed=5, Make="Yugo" },
   new Car{ PetName="Melvin", Color="White", Speed=43, Make="Ford" }
};

var query = from c in cars
            where c.Speed > 90 && c.Make == "BMW"
            select c;
```

LINQ and Nongeneric Collections

- Nongeneric collections lack the **`IEnumerable<T>`** infrastructure for querying
- This can be provided using the **`OfType<T>`** extension method

```
ArrayList cars = new ArrayList() {
   new Car{ PetName="Henry", Color="Silver", Speed=100, Make="BMW" },
   new Car{ PetName="Daisy", Color="Tan", Speed=90, Make="BMW" },
   new Car{ PetName="Mary", Color="Black", Speed=55, Make="VW" },
   new Car{ PetName="Clunker", Color="Rust", Speed=5, Make="Yugo" },
   new Car{ PetName="Melvin", Color="White", Speed=43, Make="Ford" }
};

IEnumerable<Car> enumerableCars = cars.OfType<Car>();
var query = from c in enumerableCars
            where c.Speed > 90 && c.Make == "BMW"
            select c;
```

Agenda

- Extension Methods
- Anonymous Types
- Introducing LINQ
- First Look at LINQ Query Expressions
- LINQ Query Keywords
- More Query Operator Methods
- Query Variations

The `from` Clause

- Range variables and data source are specified in the `from` clause

```
Stack<int> stack = new Stack<int>( new int[]{ 42, 87, 112, 255} );
var query = from i in stack where i < 10 select i;
```

- It can define the type of the range variable as well

```
ArrayList cars = new ArrayList {
   new Car{ PetName="Henry", Color="Silver", Speed=100, Make="BMW" },
   ...
};
var query = from Car c in cars
            where c.Speed > 90 && c.Make == "BMW"
            select c;
```

- Can in fact have multiple `from` clauses…

The **where** Clause

- Filtering conditions are specified by a boolean expression in a **where** clause

```
List<Car> cars = new List<Car> {
   new Car{ PetName="Henry", Color="Silver", Speed=100, Make="BMW" },
   ...
};
var query = from c in cars
            where c.Speed > 90 && c.Make == "BMW"
            select c;
```

```
var query = from c in cars
            where c.Speed > 90
            where SomePredicate( c )
            select c;
```

- Can have multiple **where** clauses also

The **select** Clause

- Projections of results are done through the **select** clause

```
List<Car> cars = new List<Car> {
   new Car{ PetName="Henry", Color="Silver", Speed=100, Make="BMW" },
   ...
};
var query = from c in cars
            where c.Speed > 90 && c.Make == "BMW"
            select c.Make;
```

```
var query = from c in cars
            where c.Speed > 90 && c.Make == "BMW"
            select new { c.Make, c.Color };
```

- Projections can create new (anonymous) data types

The **orderby** Clause

- Results can be sorted using the **orderby** clause

```
List<Car> cars = new List<Car> {
   new Car{ PetName="Henry", Color="Silver", Speed=100, Make="BMW" },
   ...
};
var query = from c in cars
            where c.Speed >= 55
            orderby c.PetName
            select c;
```

- The order can be **ascending** (the default) or **descending**

```
var query = from c in cars
            where c.Speed >= 55
            orderby c.PetName descending, c.Color
            select c;
```

Query Operators Resolution

TEKNOLOGISK INSTITUT

- These query operators are keywords with syntax highlighting and IntelliSense
- But they are resolved as extension methods in the **`Enumerable`** class

```
var query = from g in wiiGames
            where g.Length >= 18
            select g;
```

```
var query = wiiGames.Where( g => g.Length >= 18 )
                    .OrderBy( g => g )
                    .Select( g => g );
```

- You can use either syntax or use delegates instead of anonymous methods etc.

Agenda

- Extension Methods
- Anonymous Types
- Introducing LINQ
- First Look at LINQ Query Expressions
- LINQ Query Keywords
- **More Query Operator Methods**
- Query Variations

Count<T>

- You can compute the number of items in the result set with **Count<T>**

```
string[] wiiGames = {
   "Super Mario Galaxy",
   "FIFA 09",
   "Guitar Hero III",
   "Wii Sports",
   "Wii Fit",
   "Legend of Zelda: Twilight Princess"
};
var query = from g in wiiGames
            where g.Length >= 18
            select g;
Console.WriteLine( "{0} games match the query", query.Count() );
```

- This forces an evaluation of the query expression!

Reverse<T>

- You can reverse the result sequence with **Reverse<T>**

```
string[] wiiGames = {
   "Super Mario Galaxy",
   ...
};

var query = ( from g in wiiGames select g ).Reverse();
```

- Note that this does not evaluate the query expression...!

Set Operations: **Except<T>**

- Differences between queries can be computed with **`Except<T>`**

```
string[] wiiGames = {
   "Super Mario Galaxy", ...
};
string[] xbox360Games = {
   "Halo", ...
};

var query = ( from g in wiiGames select g ).Except(
   from g in xbox360Games select g );
var query2 = wiiGames.Except( xbox360Games );
```

- Do you think this will evaluate the query expression? ☺
- **`Union<T>`**, **`Intersect<T>`**, and **`Except<T>`** constitute the set operations (**`Distinct<T>`** is also helpful!)

Singleton Operations

- A single element can be retrieved from a query result
  - `First<T>`
  - `Last<T>`
  - `Single<T>`

```
var query = wiiGames.Intersect( xbox360Games );

var first = query.First();
var last = query.Last();
var theOnlyOne = query.Single();

Console.WriteLine( first );
Console.WriteLine( last );
Console.WriteLine( theOnlyOne );
```

- Each of these has an …`OrDefault<T>` version
  - `FirstOrDefault<T>`
  - `LastOrDefault<T>`
  - `SingleOrDefault<T>`

Partitioning Operators

- Take() and Skip()

```
string[] wiiGames = {
   "Super Mario Galaxy", ...
};
string[] xbox360Games = {
   "Halo", ...
};

var query1 = wiiGames.Union( xbox360Games ).Take( 7 );
var query2 = wiiGames.Union( xbox360Games ).Skip( 3 );
```

- There are also
  - TakeWhile()
  - SkipWhile()

Aggregation Operators

- **Aggregate()** computes a running value

```
int[] numbers = { 42, 87, 112, 176, 255 };
var result = numbers.Aggregate( ( product, i ) => product * i );

Console.WriteLine( "The product of numbers is " + result );
```

- Other aggregation operators include
  - Count()
  - Sum()
  - Min()
  - Max()
  - Average()

Agenda

- Extension Methods
- Anonymous Types
- Introducing LINQ
- First Look at LINQ Query Expressions
- LINQ Query Keywords
- More Query Operator Methods
- **Query Variations**

Grouping

- Use the **`group`** keyword or the **`GroupBy()`** method
  - Resulting query yields a set of keyed result groups

```
var query = from i in numbers
            group i by i % 2;

foreach ( var group in query )
{
   Console.WriteLine( group.Key );
   foreach ( var i in group )
   {
      Console.WriteLine( "\t" + i );
   }
}
```

- There is also a more sophisticated **`group into`** syntax

Joins

- Use the join keyword to join elements on equality

```
var query = from c in customers
            join o in orders on c.Id equals o.CustomerId
            select new
            {
               Name = c.Name,
               Product = o.Product
            };
foreach ( var cop in query )
{
   Console.WriteLine( "{0} bought {1}", cop.Name,
                                        cop.Product.Name );
}
```

- Other variations of join can be expressed in a number of ways…

Quiz: LINQ Query Expressions – Right or Wrong?

✗

```
int[] numbers = {10, 20, 30, 40, 1, 2, 3, 8};
var query = from i in numbers where i select i;
```

✓

```
List<Car> cars = new List<Car>() {
   new Car{ PetName="Henry", Color="Red", Make="BMW", Speed=55},
   ...
};
var query = from c in cars
            where 40 <= c.Speed && c.Speed < 90
            where c.Make.StartsWith( "v" )
            select c.Color;
```

✗

```
string[] wiiGames = { ... };
string[] xbox360Games = { ... };
var query = ( from g in wiiGames select g ).Except<Game>(
   from g in xbox360Games select g );
```

✓

```
var query = from c in cars
            orderby c.PetName ascending, c.Speed descending
            select new { Name = c.PetName, Model = c.Make };
```

Summary

- Extension Methods
- Anonymous Types
- Introducing LINQ
- First Look at LINQ Query Expressions
- LINQ Query Keywords
- More Query Operator Methods
- Query Variations

Question 1

```

{
   public static bool IsPrime(                    )
   {
      if ((number % 2) == 0) { number == 2; }
      int sqrt = (int) Math.Sqrt(number);

      for (int t = 3; t <= sqrt; t = t + 2)
      {
         if (number % t == 0) { return false; }
      }
      return number != 1;
   }
}
```

- You need to ensure the MyExtensions class implements the IsPrime() method on integers. Drag appropriate code segments to the correct locations?

- public class MyExtensions
- static class MyExtensions
- protected static class MyExtensions
- this int number
- int number
- this number

Question 2

- You are creating an application calling a method, which returns an array of integers named customerIds. You declare and initialize an integer variable named idToRemove. You declare an array named filteredCustomerIds. You must meet the following requirements:
  - Sort the array from highest value to lowest value
  - Remove duplicate integers from the customerIds array
  - Remove the idToRemove value from the customerIds array.
- Which code segment should you use?

a)

```
filteredCustomerIds = customerIds.Distinct()
    .OrderByDescending( i => i ).ToArray();
```

[✓ b)]

```
filteredCustomerIds = customerIds.Distinct()
    .Where( i => i != idToRemove )
    .OrderByDescending( i => i ).ToArray();
```

c)

```
filteredCustomerIds = customerIds
    .Where( i => i != idToRemove )
    .OrderByDescending( i => i ).ToArray();
```

d)

```
filteredCustomerIds = customerIds.Distinct()
    .Where( i => i != idToRemove )
    .OrderBy ( i => i ).ToArray();
```

TEKNOLOGISK
INSTITUT