WEB524

WEB PROGRAMMING ON WINDOWS

WEEK 3 – LECTURE 2

LANGUAGE INTEGRATED QUERY

Language Integrated Query (LINQ)

- As you have learned, we will most often use LINQ for *query* tasks:
  - locating/selecting one item (e.g. “find”, “single or default”)
  - filtering a collection to select items which match a condition (e.g. “where”)
  - sorting (ordering) a collection (e.g. “order by”)

- Let’s look at all of these, in a bit more detail.

Fetching One Matching Object

- You have already seen and used the Find() method
- This method only works when the following *two conditions* are true:
  - You are working with a **DbSet<TEntity>**
  - You are working with the **primary key** property. *What are the three methods of specifying a primary key on a design model class?*

Seneca Student Example

- Consider a situation where we have a Seneca Student entity.
- Each student has (at least) three properties which will have unique values:
  - A unique identifier (primary key such as a SQL int identity column)
  - Seneca Student Identification Number (Student ID)
  - Canada Social Insurance Number (SIN).

Can you use Find() to locate an object by Student ID or SIN?

Seneca Student Example (con't)

- You **cannot** use Find() to locate a student using the Student ID or SIN?
- Why?

- Neither property refers to the primary key. The Find() method takes an argument that is the primary key value.

Seneca Student Example (solution)

- We can use the SingleOrDefault() method.
- It requires a **lambda expression** as an argument.

```
// The Find() method, when we can use the primary key to locate the
object
var student = ds.Students.Find(123);

// The SingleOrDefault() method, when we need to use a lambda expression
var student = ds.Students.SingleOrDefault(s => s.SIN == “123456”);
```

Find()

- Reminder: this method only works when the following *two conditions* are true:
  - You are working with a **DbSet<TEntity>**
  - You are working with the **primary key** property.
- The Find() method is provided by the Entity Framework (it is not a typical LINQ method).
- Why bother duplicating the SingleOrDefault() method?
  - The Find() method will first check an in-memory temporary workspace for the object matching the primary key.
  - If the object is not already in-memory, a database query is performed to find the object.
- Both Find() and SingleOrDefault() may return **null**.

Similar Methods

- There are several extension methods that a beginner may view as similar:
  - Find – *only works with a DbSet<TEntity>*
  - First – *returns the first match, if no match found then it throws an exception*
  - FirstOrDefault – *returns the first match, if no match found returns null*
  - Single – *returns the only match, if no match found or more than one match found, it throws an exception*
  - SingleOrDefault – *returns the only match, if no match found returns null, if more than one match found then it throws an exception*
- We will use SingleOrDefault() if we are not searching on a primary key or we are not using a DbSet<TEntity>. It most closely resembles the Find() method.

Filtering (Where)

- Before returning a set of results, we often need to perform other tasks on a collection. These tasks may include filtering or sorting.
- This is usually done from within the **Manager** class.
- Example: assume that we wish to select only the **Program** objects that have a **Credential** property value of "*Diploma*"

```
var diplomaPrograms = ds.Programs.Where(p => p.Credential == "Diploma");
```

- Remember:
  - "p" is the range *variable*. It represents *an object* in the *Programs* collection. Its type is *Program*.
  - The letter "p" does not have any other special significance - it is simply the name of a variable.
  - Like other variables, its name should be meaningful to you and other developers.

Sorting (OrderBy and ThenBy)

- Example: assume that we wish to sort the *Program* collection by a property named *Code*

```
var sortedPrograms = ds.Programs.OrderBy(p => p.Code);
```

- The OrderBy extension method is defined in the Queryable class.
- OrderBy can be used on any object of type IQueryable<T> and it will return an IOrderedQueryable<T>.
- If you need to sort on a second (or third) property, use additional ThenBy() methods. ThenBy() uses the same style of lambda expression.
- Both OrderBy() and ThenBy() will sort in ascending order. Each has a different variant that will sort in reverse order, namely: OrderByDescending() and ThenByDescending().

Filtering and Sorting

- You can combine tasks by *chaining* the filtering and sorting tasks.
- You should first filter the collection and then sort it. Why?
- Example using the "fluent" syntax:

```
var c = ds.Programs
          .Where(p => p.Credential == "Degree"))
          .OrderBy(p => p.Code);
```

- In English: Get each program in the **Programs** collection where the Program's **Credential** property matches the string "*Degree*" then sort the results by the Program's **Code** property value.

Return Types

- In the Manager class, many of the methods will return the results of a query.
- What return type must be used? Usually, a type that's based on a view model class or a C# type (such as int or string)
- Do NOT return results directly from the data context!

Return Types (Examples)

- For a single object, return an object based on a view model class. You can use AutoMapper to help with this.

```
return mapper.Map<Supplier, SupplierBaseViewModel>(supplier);
```

- When returning a collection, return an enumerable list that's based on a view model class. Again, you can use AutoMapper to help you.

```
return mapper.Map<IEnumerable<Program>, IEnumerable<ProgramBaseViewModel>>(prog
                                                                        rams);
```

Deferred Execution

- You should be aware that your query is executed when the results are used.
- The statement that uses the data context's entity set does not necessarily cause a fetch/query at the data store.
- Often, the query gets executed only when the fetch/query result is used.
- In the previous examples, the query is executed when the result is converted (by the AutoMapper statement) into a view model class or collection.
- This concept is referred to as "deferred execution".
- Misunderstanding this concept may trick new programmers into thinking they are seeing an odd behaviour or even a bug in .NET.