Asynchonous Programming and LINQ

Contents

Asynchronous Programming and LINQ

Asynchronous Programming

- This section introduces the latest approach with .NET for asynchronous programming:
  - Introduction to Asynchronous Calls
  - Task Asynchronous Pattern
  - Async Method
  - Async Event Handlers
  - Reading From File
  - Async Method Calls

Introduction to Asynchronous Calls

- Asynchronous calls allow execution of code within a separate threads, without explicitly creating a thread
- Asynchronous calls can be made using a number of patterns:
  - Asynchronous Programming Model (APM)
    - Use Begin… End… methods
  - Event Asynchronous Pattern (EAP)
  - Task Asynchronous Pattern (TAP) (.NET 4.5)
    - Latest and Simplest
    - Currently preferred
    - Many standard APIs provide support for Async calls (where appropriate)

Task Asynchronous Pattern (TAP)

- Many Methods within .NET Framework/Core now support TAP
  - File handling
  - Service Calls
  - DataProvider Command calls
- Methods typically of form …Async(…)
  - Call using new **await** keyword
  - **await** can only be called from method with **async** keywork

Async Method

- Defining Async Method:

```
private Task<string> GetMessageAsync()
{
  return Task.Run<string>(() =>
  {
    return string.Format("Manage Thread Id {0}",
            Thread.CurrentThread.ManagedThreadId);
  });
}
```

Convention For Async Methods

- Returned Task should be ‘hot’ (running/started)

Async Event Handlers

- Event handlers can be ‘async’
  - In Graphical User interfaces
- Event handlers return void
  - This is one of few occasions for using async on method with void return type

```
private async void ButtonMessage_Click(object sender,
                                       RoutedEventArgs e)
{
  ListBoxOutput.Items.Add( await GetMessageAsync());
}
```

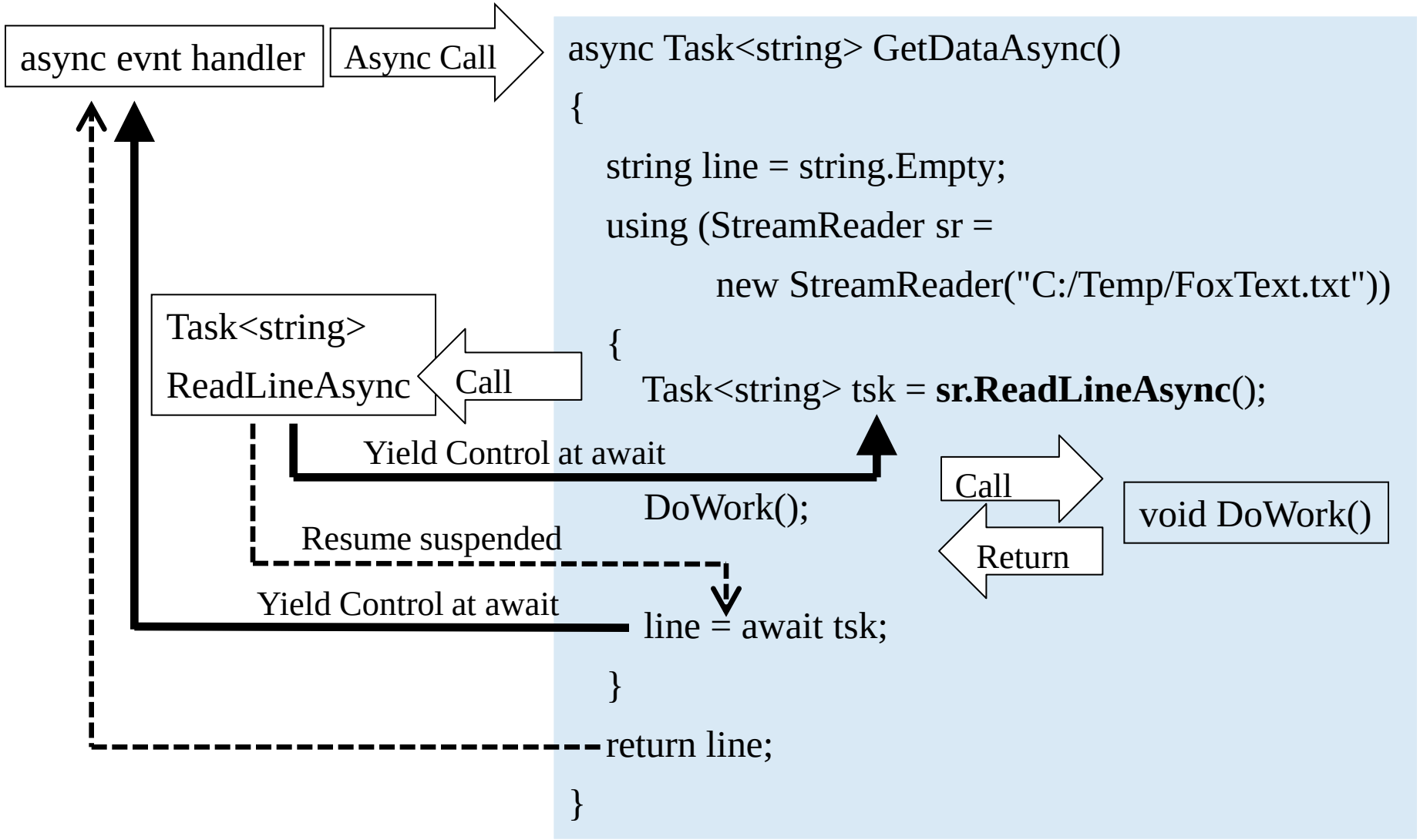
Reading From File

- Multiple awaits can take place within method:

```
async Task<string> GetDataAsync()
{
  string line = string.Empty;

  using (StreamReader sr =
     new StreamReader("C:/Temp/FoxText.txt"))
  {
    while (!sr.EndOfStream)
    {
      line += await sr.ReadLineAsync();
    }
  }
  return line;
}
```

Async Method Calls



Asynchronous Programming - Summary

- This section introduced the latest approach with .NET for asynchronous programming:
  - Introduction to Asynchronous Calls
  - Task Asynchronous Pattern
  - Async Method
  - Async Event Handlers
  - Reading From File
  - Async Method Calls

Lambda Expressions and Anonymous Types

- This Section will give an overview of:
  - Introduction
  - Lambda Expressions
  - Extension Methods
  - Implicitly Typed Variables
  - Anonymous Types

Introduction

- Independently some of these features may not appear very useful!
  - Become important in conjunction with framework features
  - Needed for new LINQ capabilities
  - Some build on existing features
    - Anonymous methods – Lambda Expressions

Using Generic Delegates (Named) (C#)

- Generic Delegates can be used to declare reference variables for methods:

```
class Program
{
  public static int Square(int a)
  {
    return a * a;
  }
}
```

- Using conventional Methods:

```
Func<int, int> calc = Program.Square;
```

Generic Delegate

Lambda Expressions (C#)

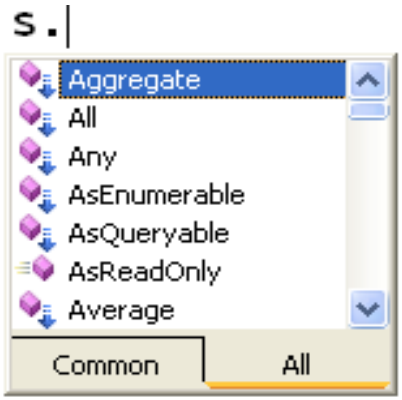
- Lambda Expressions are an extension of Anonymous Methods
  - Lambda Expressions allow Functional Programming
    - Inputs are processed to return a result
    - No side effects!
- C# allows a number of ways of defining Lambda Expressions:

```
Func<int, int> f1 = (int a) => a * a;
// or
Func<int, int> f2 = (a) => a * a;
// or
Func<int, int> f3 = a => a * a;
```

Type is Inferred From Usage

Predefined Extension Methods

- Many predefined Extension Methods, including:
  - Aggregate
  - All
  - Any
  - OrderBy
  - Select
  - Where



- Many of these take Lambda expressions

Defining Extension Methods (C#)

```
public static class MyExtensions
{
  public static int Maximum(this int[] ta)
  {
    int max = int.MinValue;

    foreach (int i in ta)
    {
      if (i > max) max = i;
    }
    return max;
  }
}
```

Target Type

Implicitly Typed Variables

- Implicit Typing introduces the ability to declare variables without specifying type
  - Type of variable is not object or ‘variant’
  - Type safety is preserved
- Implicit Typing only applies to local variables:

Variables are of the ‘correct’ type!

var keyword used to declare variable

```
var intVal = 3;                    // int
var longVal = 5678L;               // long
var stringVal = "Hello";           // string
```

Anonymous Types (C#)

- C# 2.0 introduced Anonymous Methods
  - C# 3.0 introduces Anonymous Types
- Anonymous types are not given a name
  - (or rather, unknown name)
  - New type can be inferred from usage, e.g.

Property Name

Property Type Inferred

```
var newData = new { ID = 1, Name = "Fred" };
var newData2 = new { ID = 2, Name = "Jim"};


newData2 = newData;
```

Same Anonymous Type!

Assignment Possible as of Same Type

Lambda Expressions and Extensions Methods - Summary

- This Section gave an overview of the new features:
  - Introduction to New Features
  - Lambda Expressions
  - Extension Methods
  - Implicitly Typed Variables
  - Anonymous Types

LINQ and Entity Framework

- This section will give an introduction to Language INtegrated Queries:
  - Introduction to Standard Query Operators
  - LINQ vs Extension Methods
  - LINQ Statement
  - Entity Framework - Background
  - Context and Querying

Introduction to Standard Query Operators

- Standard Query Operators is an API for manipulating Sequences
  - Within System.Linq namespace
  - Defined as Extension methods to act on IEnumerable<> interface
  - Sequences can be manipulated using Lambda expression in conjunction with Query Operators
  - Extensions to the C#/VB.NET Languages are designed to make the usage ‘easier’
    - SQL Like language extensions

LINQ vs Extension Methods (C#)

```
SomeData[] data = {
      new SomeData {Name = "Fred", Id = 4},
      new SomeData {Name = "Jim", Id = 7},
      new SomeData {Name = "Alice", Id = 9}};
```

Sort Data using Extension Method

Lambda Expression

```
var sortedData = data.OrderBy(o => o.Name);
```

Sort Data using LINQ Expression

```
var result = from o in data orderby o.Name select o;
```

Collection

Order by Property

LINQ Statement (C#)

- C# has been extended to include SQL like keywords:

```
int[] data = { 543, 6, 3, 56, 765 };

var result = from i in data
             where i > 100
             orderby i descending select i;

foreach (int i in result)
{
    Console.Write(" " + i);
}
```

Collection

Where Clause

Select Expression

Iterate over Results

Entity Framework – General Background

- The Entity Framework has evolved considerably
  - Currently up to EF 6 (Framework) and EF 7 (Core)
  - Entity Data Modelling visual tools no longer supported!
  - Three approaches to use of EF
    - Database First (Tooling depends on version) (EF 7 command line)
    - Model First (Not .NET Core)
    - Code First
  - Updates installed using Nuget Package Manager

Context and Querying (C#)

- Context represents connection database:

```
using(NORTHWNDEntities ctx = new NORTHWNDEntities())
{
  var custs = from c in ctx.Customers
                  where c.City == "London"
                  orderby c.CustomerID
                  select new {ID = c.CustomerID,
                              Name = c.ContactName};

  foreach(var c in custs)
  {
    Console.WriteLine(c. ID + " " + c.Name);
  }
}
```

Anonymous type

Iterate over Sequence of Customer

LINQ - Summary

- This section gave an introduction to:
  - Introduction to Standard Query Operators
  - LINQ vs Extension Methods
  - LINQ Statement
  - Entity Framework - Background
  - Context and Querying