

Session 12: Advanced Concepts in C#

For Aptech Centre Use Only

Objectives

- **Describe anonymous methods**
- **Define extension methods**
- **Explain anonymous types**
- **Explain partial types**
- **Explain nullable types**
- **Describe system-defined generic delegates**
- **Define lambda expressions**
- **Explain query expressions**
- **Explain parallel and dynamic programming**
- **Describe Entity Framework (EF) Core and Entity Framework 6**

Anonymous Methods and Their Features

- ▶ An anonymous method is an inline nameless block of code that can be passed as a delegate parameter.
- ▶ Delegates can invoke one or more named methods that are included while declaring the delegates.

Features

Inline code in the delegate declaration

Best suited for small blocks

Can accept parameters of any type

Parameters using the `ref` and `out` keywords can be passed to it

Can include parameters of a generic type

Cannot include jump statements such as `goto` and `break` that transfer control out of the scope of the method

Creating Anonymous Methods

- ▶ An anonymous method is created when you instantiate or reference a delegate with a block of unnamed code.

▶ Syntax:

```
// Create a delegate instance

<access modifier> delegate <return type>
<DelegateName> (parameters);

// Instantiate the delegate using an anonymous method

<DelegateName> <objDelegate> = new <DelegateName>
(parameters)
{ /* ... */ };
```

Snippet

```
using System;
class AnonymousMethods
{
    //This line remains same even if named methods are used
    delegate void Display();
    static void Main(string[] args)
    {
        //Here is where a difference occurs when using
        // anonymous methods
        Display objDisp = delegate()
        {
            Console.WriteLine("This illustrates an anonymous method");
        };
        objDisp();
    }
}
```

Referencing Multiple Anonymous Methods

Snippet

```
using System;
class MultipleAnonymousMethods
{
    delegate void Display();
    static void Main(string[] args)
    {
        //delegate instantiated with one anonymous
        // method reference
        Display objDisp = delegate()
        {
            Console.WriteLine("This illustrates one anonymous
                               method");
        };
        //delegate instantiated with another anonymous method
        // reference
        objDisp += delegate()
        {
            Console.WriteLine("This illustrates another anonymous
                               method with the same delegate instance");
        };
        objDisp();
    }
}
```

Output

```
This illustrates one anonymous method
This illustrates another anonymous method with
the same delegate instance
```

Passing Parameters

- ▶ The block of code within the anonymous method can access these specified parameters just like any normal method.
- ▶ You can pass the parameter values to the anonymous method while invoking the delegate.

Snippet

```
using System;
class Parameters
{
    delegate void Display(string msg, int num);
    static void Main(string[] args)
    {
        Display objDisp = delegate(string msg, int num)
        {
            Console.WriteLine(msg + num);
        };
        objDisp("This illustrates passing parameters to
        anonymous methods. The int parameter passed is: ", 100);
    }
}
```

Output

This illustrates passing parameters to anonymous methods. The int parameter passed is: 100

Extension Methods

- ▶ Extension methods allow you to extend an existing type with new functionality without directly modifying those types.
- ▶ Extension methods are static methods that have to be declared in a static class.

Snippet

```
using System;
/// <summary>
/// Class ExtensionExample defines the extension method
/// </summary>
static class ExtensionExample{
    // Extension Method to convert the first character to
    //lowercase
    public static string FirstLetterLower(this string result)
    {
        if (result.Length > 0){
            char[] s = result.ToCharArray();
            s[0] = char.ToLower(s[0]);
            return new string(s);
        }
        return result;
    }
}
class Program
{
    public static void Main(string[] args)
    {
        string country = "Great Britain";
        // Calling the extension method
        Console.WriteLine(country.FirstLetterLower());
    }
}
```

Anonymous Types

- ▶ Anonymous type:
 - ▶ Is basically a class with no name and is not explicitly defined in code.
 - ▶ Uses object initializers to initialize properties and fields. Since it has no name, you must declare an implicitly typed variable to refer to it.

Code to understand the use of anonymous types:

Snippet

```
using System;
class AnonymousTypeExample
{
    public static void Main(string[] args)
    {
        // Anonymous Type with three properties.
        var stock = new { Name = "Michigan Enterprises", Code = 1301,
                        Price = 35056.75 };
        Console.WriteLine("Stock Name: " + stock.Name);
        Console.WriteLine("Stock Code: " + stock.Code);
        Console.WriteLine("Stock Price: " + stock.Price);
    }
}
```

- ▶ When an anonymous type is created, the C# compiler carries out the following tasks:
 - ◆ Interprets the type
 - ◆ Generates a new class
 - ◆ Use the new class to instantiate a new object
 - ◆ Assigns the object with the required parameters
- ▶ The compiler internally creates a class with the respective properties when code is compiled.
- ▶ In this program, the class might look like the one that is shown in code.

Partial Types and Their Features

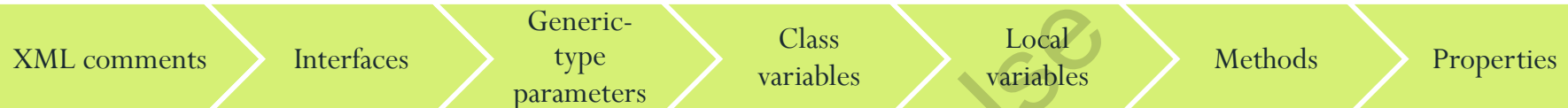
- ▶ When you have C# class or structure with lots of member definitions, split the data members of the class or structure and store them in different files.
- ▶ These members can be combined into a single unit while executing the program.
- ▶ This can be done by creating partial types.
- ▶ The partial types feature facilitates the definition of classes, structures, and interfaces over multiple files.

Benefits			
Separate the generator code from the application code	Help in easier development and maintenance of the code	Make the debugging process easier	Prevent programmers from accidentally modifying the existing code

Merged Elements during Compilation 1-2

- ▶ The members of partial classes, partial structures, or partial interfaces declared and stored at different locations are combined together at the time of compilation.

Members can include:



- ▶ A partial type can be compiled at the Developer Command Prompt for VS2019. The command to compile a partial type is:

```
csc /out:<FileName>.exe <CSharpFileNameOne>.cs <CSharpFileNameTwo>.cs
```

where,

FileName: Is the user specified name of the .exe file.

CSharpFileNameOne: Is the name of the first file where a partial type is defined.

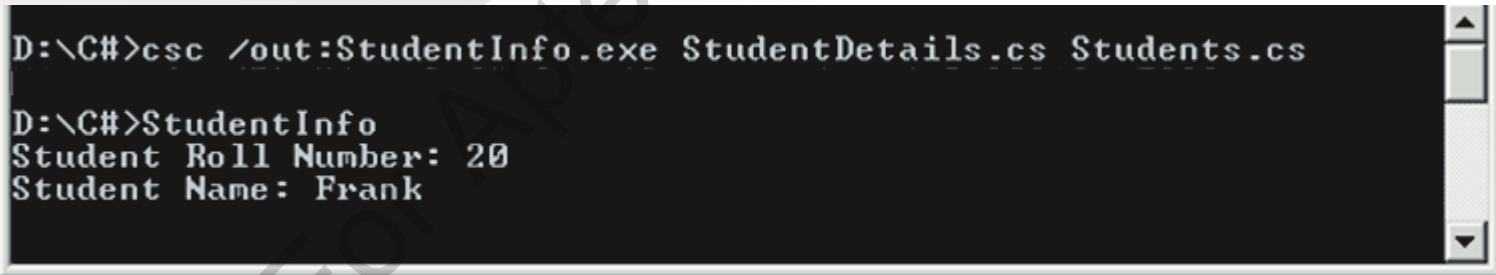
CSharpFileNameTwo: Is the name of the second file where a partial type is defined.

Snippet

```
using System;
using System.Collections.Generic;
using System.Text;
//Stored in StudentDetails.cs file
namespace School {
    public partial class StudentDetails {
        int _rollNo;
        string _studName;
        public StudentDetails(int number, string name)
        {
            _rollNo = number;
            _studName = name;
        }
    }
}
```

Merged Elements during Compilation 2-2

```
using System;
using System;
using System.Collections.Generic;
using System.Text;
//Stored in Students.cs file
namespace School {
    public partial class StudentDetails {
        public void Display() {
            Console.WriteLine("Student Roll Number: " + _rollNo);
            Console.WriteLine("Student Name: " + _studName);
        }
    }
    public class Students {
        static void Main(string[] args) {
            StudentDetails objStudents = new StudentDetails(20,
                "Frank");
            objStudents.Display();
        }
    }
}
```



```
D:\C#>csc /out:StudentInfo.exe StudentDetails.cs Students.cs

D:\C#>StudentInfo
Student Roll Number: 20
Student Name: Frank
```

Implementing Partial Types (1-3)

- ▶ `partial` keyword:
 - Specifies that the code is split into multiple parts and these parts are defined in different files and namespaces.
- ▶ The type names of all the constituent parts of a partial code are prefixed with the `partial` keyword.
 - For example, if the complete definition of a structure is split over three files, each file must contain a partial structure having the partial keyword preceding the type name.

Syntax

```
[<access modifier>] [keyword] partial <type>  
<Identifier>
```

where,

- ▶ **access_modifier:** Optional access modifier such as `public`, `private`, and so on.
- ▶ **keyword:** Optional keyword such as `abstract`, `sealed`, and so on.
- ▶ **type:** A specification for a class, a structure, or an interface.
- ▶ **Identifier:** The name of the class, structure, or an interface.

Implementing Partial Types (2-3)

Code to create an interface with two partial interface definitions:

Snippet

```
using System;
//Program Name: MathsDemo.cs
partial interface MathsDemo{
    int Addition(int valOne, int valTwo);
}
//Program Name: MathsDemo2.cs
partial interface MathsDemo{
    int Subtraction(int valOne, int valTwo);
}
class Calculation : MathsDemo{
    public int Addition(int valOne, int valTwo) {
        return valOne + valTwo;
    }
    public int Subtraction(int valOne, int valTwo) {
        return valOne - valTwo;
    }
    static void Main(string[] args) {
        int numOne = 45;
        int numTwo = 10;
        Calculation objCalculate = new Calculation();
        Console.WriteLine("Addition of two numbers: " +
            objCalculate.Addition(numOne, numTwo));
        Console.WriteLine("Subtraction of two numbers: " +
            objCalculate.Subtraction(numOne, numTwo));
    }
}
```

Implementing Partial Types (3-3)

R

The partial-type definitions must include the partial keyword in each file.

U

The partial keyword must always follow the class, struct, or interface keywords.

L

The partial-type definitions of the same type must be saved in the same assembly.

E

The partial-type definitions of the same type must be saved in the same assembly.

S

- ▶ The partial-type definitions can contain certain C# keywords which must exist in the declaration in different files. These keywords are as follows:

public

private

protected

internal

abstract

sealed

new

Output

Addition of two numbers: 55

Subtraction of two numbers: 35

Partial Classes (1-2)

- ▶ A class is one of the types in C# that supports partial definitions
- ▶ Classes can be defined over multiple locations to store different members such as variables, methods, and so on
- ▶ Definitions are combined during compilation to create a single class

Snippet

```
using System;
//Program: StudentDetails.cs
public partial class StudentDetails {
    public void Display() {
        Console.WriteLine("Student Roll Number: " + _rollNo);
        Console.WriteLine("Student Name: " + _studName);
    }
}

//Program StudentDetails2.cs
public partial class StudentDetails {
    int _rollNo;
    string _studName;
    public StudentDetails(int number, string name) {
        _rollNo = number;
        _studName = name;
    }
}

public class Students {
    static void Main(string[] args) {
        StudentDetails objStudents = new StudentDetails(20,
            "Frank");
        objStudents.Display();
    }
}
```

Partial Methods

- ▶ A partial method is a method whose signature is included in a partial type, such as a partial class or `struct`. The method may be optionally implemented in another part of the partial class or type or same.
- ▶ Partial methods are useful when part of the code has been auto-generated by a tool or IDE and the other parts of the code require customization.

For Aptech Centre Use Only

Nullable Types (1-2)

- ▶ To identify and handle value type fields with null values.
- ▶ Value type variables with null values were indicated either by using a special value or an additional variable.

Creating Nullable Types

- ▶ Indicates that a variable can have the value `null`.
- ▶ Nullable types are instances of the `System.Nullable<T>` structure.
- ▶ A variable can be made `nullable` by adding a question mark following the data type.
- ▶ Alternatively, it can be declared using the generic `Nullable<T>` structure present in the `System` namespace.
- ▶ A nullable type can include any range of values that is valid for the data type to which the nullable type belongs.
- ▶ For example, a `bool` type that is declared as a nullable type can be assigned the values `true`, `false`, or `null`.
- ▶ Nullable types have two public read-only properties that can be implemented to check the validity of nullable types and to retrieve their values.

The `HasValue` property:

- Is a `bool` property that determines validity of the value in a variable. The `HasValue` property returns a `true` if the value of the variable is `not null`, else it returns `false`.

The `Value` property:

- Identifies the value in a nullable variable. When the `HasValue` evaluates to `true`, the `Value` property returns the value of the variable, otherwise it returns an exception.

Nullable Types (2-2)

- ▶ Code to display the employee's name, ID, and role using the nullable types:

Snippet

```
using System;
class Employee
{
    static void Main(string[] args)
    {
        int empId = 10;
        string empName = "Patrick";
        char? role = null;
        Console.WriteLine("Employee ID: " + empId);
        Console.WriteLine("Employee Name: " + empName);
        if (role.HasValue == true)
        {
            Console.WriteLine("Role: " + role.Value);
        }
        else
        {
            Console.WriteLine("Role: null");
        }
    }
}
```

Output

```
Employee ID: 10
Employee Name: Patrick
Role: null
```

The ?? Operator

- ▶ A nullable type can either have a defined value or the value can be undefined.
- ▶ If a nullable type contains a null value and you assign this nullable type to a non-nullable type, the compiler generates an exception called `System.InvalidOperationException`.
- ▶ Following code demonstrates the use of ?? operator:

Snippet

```
using System;

class Salary{
    static void Main(string[] args) {
        double? actualValue = null;
        double marketValue = actualValue ?? 0.0;
        actualValue = 100.20;
        Console.WriteLine("Value: " + actualValue);
        Console.WriteLine("Market Value: " + marketValue);
    }
}
```

Output

```
Value: 100.2
Market Value: 0
```

Converting Nullable Types

- ▶ C# supports two types of conversions on nullable types:
 - Implicit conversion
 - Explicit conversion

Snippet

```
using System;
class ImplicitConversion
{
    static void Main(string[] args)
    {
        int? numOne = null;
        if (numOne.HasValue == true)
        {
            Console.WriteLine("Value of numOne before
            conversion: " + numOne);
        }
        else
        {
            Console.WriteLine("Value of numOne: null");
        }
        numOne = 20;
        Console.WriteLine("Value of numOne after implicit
        conversion: " + numOne);
    }
}
```

Output

```
Value of numOne: null
Value of numOne after implicit conversion: 20
```

Data Handling in C#

- ▶ Data handling in C# is possible with the help of files or by using a database technology.
- ▶ ADO.NET is a database technology component of the .NET Framework that is used to connect an application system and a database server.
- ▶ ADO stands for ActiveX Data Objects. The Microsoft ADO.NET technology comprises a set of classes to interact with data sources such as XML files and databases and handle data access.
- ▶ The data is then utilized in .NET applications.

ADO.NET

- ▶ ADO.NET uses XML for storing and transferring data among applications. It is an industry standard and also provides fast access of data for desktop and distributed applications.
- ▶ The main advantage of ADO.NET is its scalability and interoperability.

Features

ADO.NET is a single object-oriented API

The code and classes in ADO.NET are managed

To work in visual form, .NET offers ADO.NET components and data-bound control

To work in visual form, .NET offers ADO.NET components and data-bound control

Snippet

```
SqlConnection con = new SqlConnection("data source=.; database=Sample;
integrated security=SSPI");
SqlCommand cmd = new SqlCommand("Select * from tblProduct", con);
con.Open();
SqlDataReader rdr = cmd.ExecuteReader();
// The SqlDataReader object now contains the retrieved data
// Perform some tasks with the data
...
...
con.Close();
```

Entity Framework

- ▶ Entity Framework (EF) refers to a collection of technologies in ADO.NET that support the development of data-oriented software applications.

An open-source Object Relational Mapper (ORM) framework intended for .NET applications

Enables developers to work with data by using objects of domain-specific classes

No emphasis on the underlying database tables and columns for storing this data

Developers can achieve a high level of abstraction when working with data

Can create data-oriented applications and maintain them using lesser amount of code as compared to traditional applications

EF creates an Entity Data Model (EDM) based on Plain Old CLR Object (POCO) entities with get/set properties of various data types.

EF enables the use of LINQ queries using C# for retrieving data from the underlying database. LINQ queries are translated to the database-specific query language.

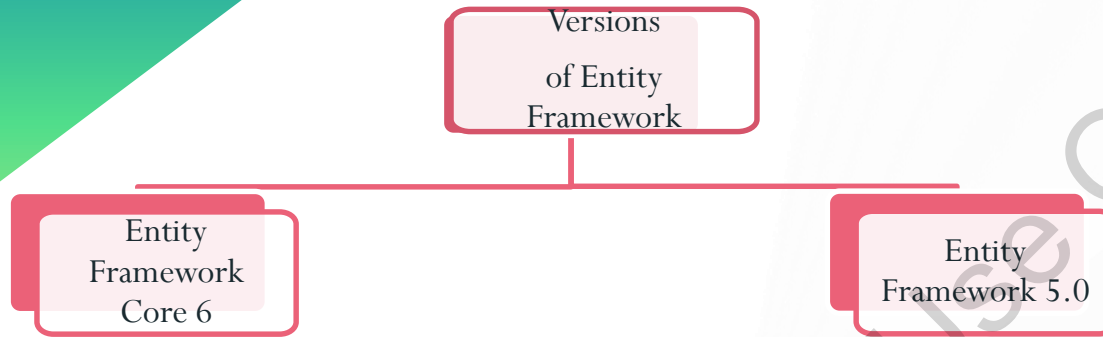
Automatic transaction management occurs when querying or saving data. EF also provides options for customizing transaction management.

EF keeps track of changes that have occurred to instances of entities (Property values) that are given to the database.

EF offers first level of caching out of the box. Thus, repeated querying will result in data returning from the cache rather than affecting the database.

Features

EF Core and EF 6



Entity Framework 6

- *Entity splitting*
- *Creating/Updating a model from the database*
- *Graphical visualization of a model*
- *Stored procedure mapping*

Entity Framework Core 5.0

- *Alternate keys*
- *Key generation*
- *SQL queries and Composing with LINQ*
- *Filtered include*

Lambda Expressions

- ▶ A method associated with a delegate is never invoked by itself, instead, it is only invoked through the delegate.
- ▶ Sometimes, it can be very cumbersome to create a separate method just so that it can be invoked through the delegate.
- ▶ To overcome this, anonymous methods and lambda expressions can be used.
- ▶ Anonymous methods allow unnamed blocks of code to be created for representing a method referred by a delegate.

Syntax

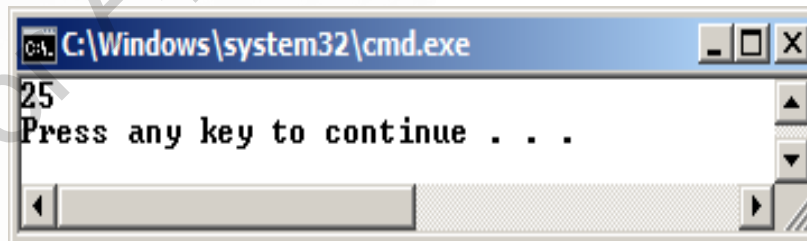
```
parameter-list => expression or statements
```

- ▶ Following code uses a lambda expression to calculate the square of an integer number:

Snippet

```
class Program {  
    delegate int ProcessNumber(int input);  
    static void Main(string[] args) {  
        ProcessNumber del = input => input * input;  
        Console.WriteLine(del(5));  
    }  
}
```

Output



Expression Lambdas (1-2)

- ▶ 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
```

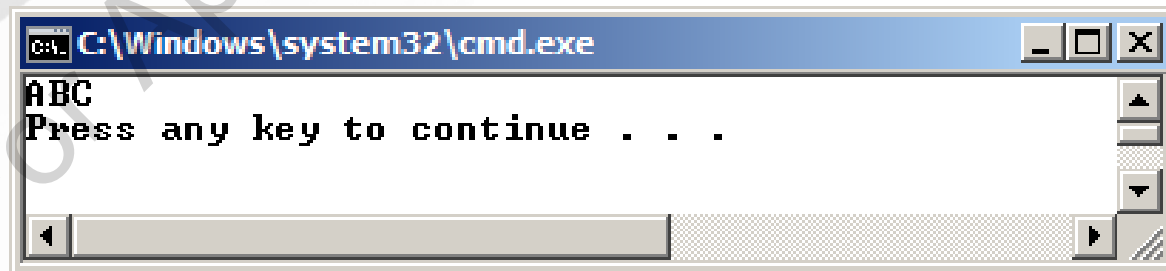
Expression Lambdas (2-2)

- ▶ 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.

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"));  
    }  
}
```

Output



Statement Lambdas

- ▶ 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,

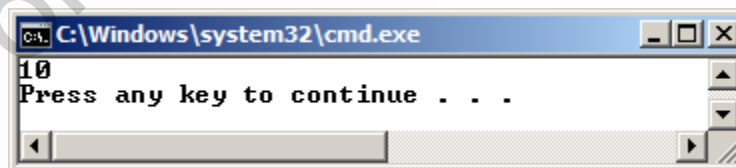
- ▶ input_parameters: one or more input parameters, each separated by a comma.
- ▶ statement: a statement body containing one or more statements.
- ▶ Optionally, you can specify a return statement to get the result of a lambda.

```
(string str, string str1)=> { return (str==str1);}
```

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");
    }
}
```

Output



Lambdas with Standard Query Operators

- ▶ Lambda expressions can also be used with standard query operators.

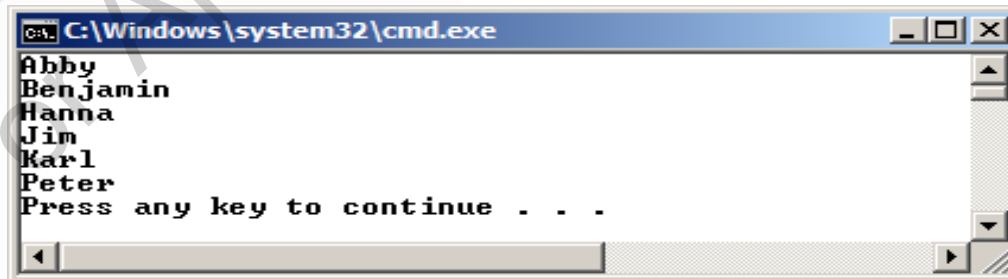
Following table lists the standard query operators:

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

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);
        }
    }
}
```

Output



```
C:\Windows\system32\cmd.exe
Abby
Benjamin
Hanna
Jim
Karl
Peter
Press any key to continue . . .
```

Query Expressions

- ▶ A query expression is a query that is written in query syntax using clauses such as from, select, and so forth. These clauses are an inherent part of a LINQ query.
- ▶ LINQ is a set of technologies introduced in Visual Studio 2008 that simplifies working with data present in various formats in different data sources. LINQ provides a consistent model to work with such data.

Snippet

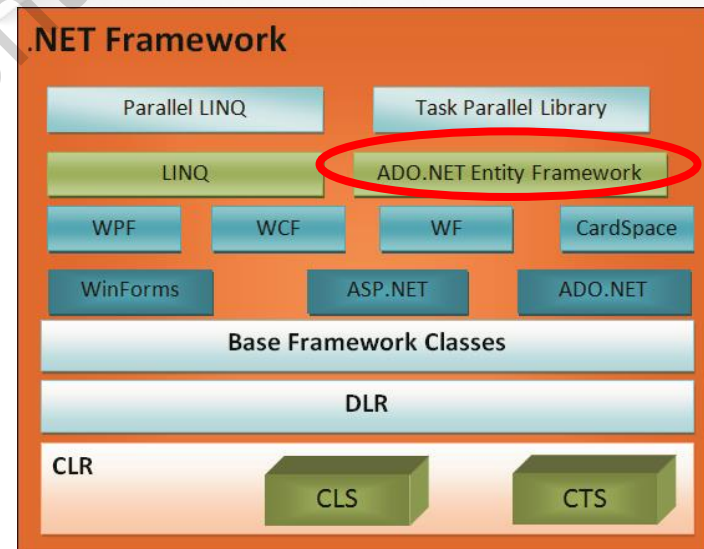
```
class Program {  
    static void Main(string[] args) {  
        string[] names = { "Hanna", "Jim", "Pearl", "Mel", "Jill",  
                           "Peter", "Karl", "Abby", "Benjamin" };  
        IEnumerable<string> words = from word in names  
                                     where word.EndsWith("l")  
                                     select word;  
  
        foreach (string s in words)  
            Console.WriteLine(s);  
    }  
}
```

- ▶ Some of the commonly used query keywords seen in query expressions are listed in the following table:

Clause	Description
from	Used to indicate a data source and a range variable
where	Used to filter source elements based on one or more boolean expressions that may be separated by the operators && or
select	Used to indicates how the elements in the returned sequence will look like when the query is executed
group	Used to group query results based on a specified key value
orderby	Used to sort query results in ascending or descending order
ascending	Used in an orderby clause to represent ascending order of sort
descending	Used in an orderby clause to represent descending order of sort

Accessing Databases Using the Entity Framework

- ▶ To address data access requirements of enterprise applications, Object Relationship Mapping (ORM) frameworks have been introduced.
- ▶ An ORM framework simplifies the process of accessing data from applications and performs the necessary conversions between incompatible type systems in relational databases and object-oriented programming languages.
- ▶ The Entity Framework is an ORM framework that .NET applications can use.

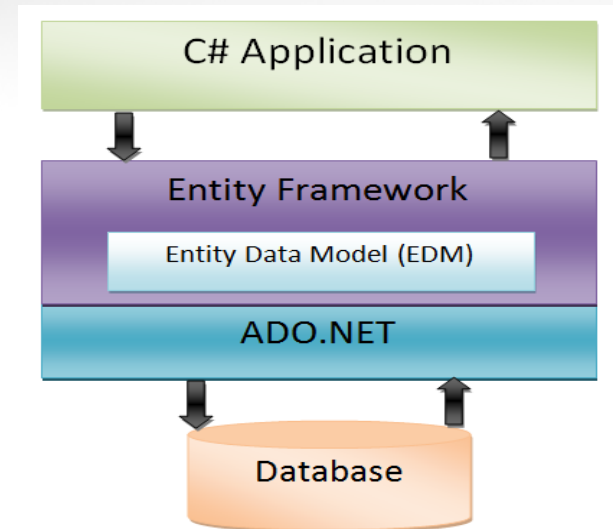


The Entity Data Model

- ▶ The Entity Framework is an implementation of the Entity Data Model (EDM), which is a conceptual model that describes the entities and the associations they participate in an application.
- ▶ EDM allows a programmer to handle data access logic by programming against entities without having to worry about the structure of the underlying data store and how to connect with it.

Example

- ▶ In an order placing operation of a customer relationship management application, a programmer using the EDM can work with the Customer and Order entities in an object-oriented manner without writing database connectivity code or SQL-based data access code.



Role of EDM in the Entity Framework architecture

Development Approaches

- ▶ Entity Framework eliminates the necessity to write most of the data-access code that otherwise must be written and uses different approaches to manage data related to an application which are as follows:

The database-first approach

- The Entity Framework creates the data model containing all the classes and properties corresponding to the existing database objects, such as tables and columns.

The model-first approach

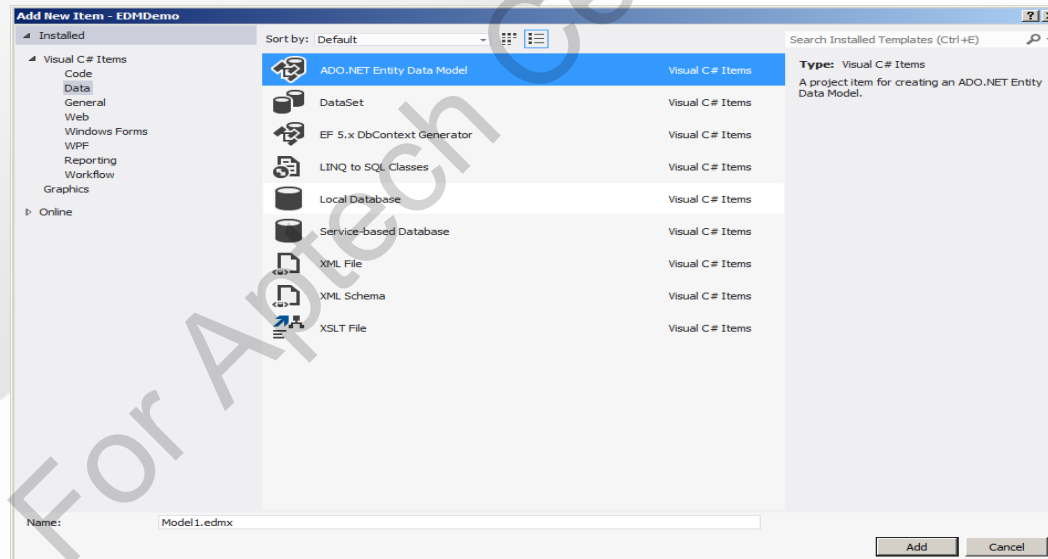
- The Entity Framework creates database objects based on the model that a programmer creates to represent the entities and their relationships in the application.

The code-first approach

- The Entity Framework creates database objects based on custom classes that a programmer creates to represent the entities and their relationships in the application.

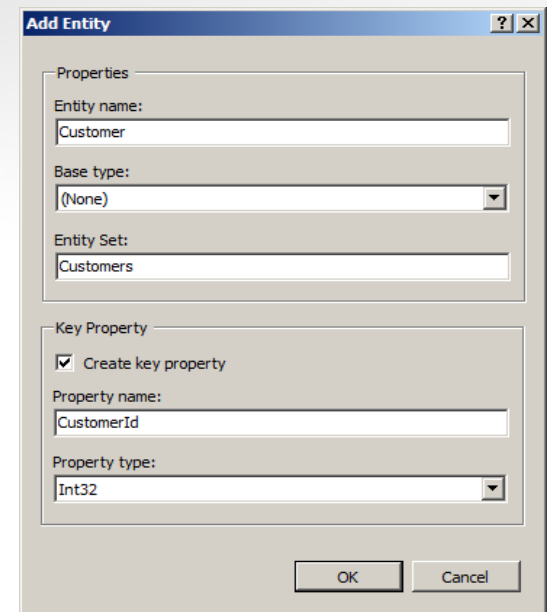
Creating an Entity Data Model (1-2)

- ▶ To create an entity data model and generate the database object, a programmer must perform the following steps:
 - ▶ Open **Visual Studio 2012**.
 - ▶ Create a Console Application project, named **EDMDemo**.
 - ▶ Right-click **EDMDemo** in the **Solution Explorer** window, and select **Add → New Item**. The **Add New Item – EDMDemo** dialog box is displayed.
 - ▶ Select **Data** from the left menu and then, select **ADO.NET Entity Data Model** as shown in the following figure:



Creating an Entity Data Model (2-2)

- ▶ Click **Add**. The **Entity Data Model Wizard** appears.
- ▶ Select **Empty model**.
- ▶ Click **Finish**. The **Entity Data Model Designer** is displayed.
- ▶ Right-click the Entity Data Model Designer, and select **Add New** → **Entity**. The **Add Entity** dialog box is displayed.
- ▶ Enter **Customer** in the **Entity name** field and **CustomerId** in the **Property name** field.
- ▶ Click **OK**. The Entity Data Model Designer displays the new Customer entity.
- ▶ Right-click the **Customer** entity and select **Add New** → **Scalarproperty**.
- ▶ Enter **Name** as the name of the property.
- ▶ Similarly, add an **Address** property to the **Customer** entity.
- ▶ Add another entity named **Order** with an **OrderId** key property.
- ▶ Add a **Cost** property to the **Order** entity.



The screenshot shows the 'Add Entity' dialog box with the following details:

- Properties**
 - Entity name: Customer
 - Base type: (None)
 - Entity Set: Customers
- Key Property**
 - ☒ Create key property
 - Property name: CustomerId
 - Property type: Int32
- Buttons: OK, Cancel

Defining Relationships

- ▶ After creating an EDM and adding the entities to the EDM, the relationships between the entities can be defined using the Entity Data Model Designer.
- ▶ As a customer can have multiple orders, the Customer entity will have a one-to-many relationship with the Order entity.
- ▶ To create an association between the Customer and Order entities:
 - ▶ Right-click the Entity Data Model Designer, and select **Add New → Association**. The **Add Association** dialog box is displayed.
 - ▶ Ensure that the left-hand **End** section of the relationship point to **Customer** with a multiplicity of 1 (**One**) and the right-side **End** section point to **Post** with a multiplicity of *(**Many**). Accept the default setting for the other fields.
 - ▶ Click **OK**. The Entity Data Model Designer displays the entities with the defined relationship.

Creating Database Objects

- ▶ After designing the model of the application, the programmer must generate the database objects based on the model. To generate the database objects:
 - ▶ Right-click the Entity Data Model Designer and select **Generate Database from Model**. The **Generate Database Wizard** dialog box appears.
 - ▶ Click **New Connection**. The **Connection Properties** dialog box is displayed.
 - ▶ Enter **(localdb)\v11.0** in the **Server name** field and **EDMDemo.CRMDB** in the **Select or enter a database name** field.
 - ▶ Click **OK**. Visual Studio will prompt whether to create a new database.
 - ▶ Click **Yes**.
 - ▶ Click **Next** in the **Generate Database Wizard** window. Visual Studio generates the scripts to create the database objects.
 - ▶ Click **Finish**. Visual Studio opens the file containing the scripts to create the database objects.
 - ▶ Right-click the file and select **Execute**. The **Connect to Server** dialog box is displayed.
 - ▶ Click **Connect**. Visual Studio creates the database objects.

Using the EDM (1-2)

- ▶ When a programmer uses Visual Studio to create an EDM with entities and their relationships, Visual Studio automatically creates several classes.
- ▶ The important classes that a programmer will use are:

Database Context Class

This class extends the `DbContext` class of the `System.Data.Entity` namespace to allow a programmer to query and save the data in the database.

In the **EDM Demo** Project, the **`Model1Container`** class present in the `Model1.Context.cs` file is the database context class.

Entity Classes

These classes represent the entities that programmers add and design in the Entity Data Model Designer.

In the EDM Demo Project, `Customer` and `Order` are the entity classes.

Using the EDM (2-2)

Snippet

```
class Program{

    static void Main(string[] args)    {
        using (Model1Container dbContext = new Model1Container()){
            Console.WriteLine("Enter Customer name: ");
            var name = Console.ReadLine();
            Console.WriteLine("Enter Customer Address: ");
            var address = Console.ReadLine();
            Console.WriteLine("Enter Order Cost:");
            var cost = Console.ReadLine();
            var customer = new Customer { Name=name,Address=address};
            var order = new Order { Cost = cost };
            customer.Orders.Add(order);
            dbContext.Customers.Add(customer);
            dbContext.SaveChanges();
            Console.WriteLine("Customer and Order Information added successfully.");
        }
    }
}
```

Output

```
Enter Customer name: Alex Parker
Enter Customer Address: 10th Park Street, Leo Mount
Enter Order Cost:575
Customer and Order Information added successfully
```

Querying Data by Using LINQ Query Expressions (1-3)

- ▶ To create and execute queries against the conceptual model of Entity Framework, programmers can use LINQ to Entities. In LINQ to Entities, a programmer creates a query that returns a collection of zero or more typed entities.
- ▶ To create a query, the programmer requires a data source against which the query will execute.
- ▶ An instance of the `ObjectQuery` class represents the data source. In LINQ to entities, a query is stored in a variable. When the query is executed, it is first converted into a command tree representation that is compatible with the Entity Framework.
- ▶ Then, the Entity Framework executes the query against the data source and returns the result.

Snippet

```
public static void DisplayPropertiesMethodBasedQuery()
public static void DisplayAllCustomers() {
    using (Model1Container dbContext = new Model1Container()) {
        IQueryable<Customer> query = from c in dbContext.Customers select c;
        Console.WriteLine("Customer Order Information:");
        foreach (var cust in query){
            Console.WriteLine("Customer ID: {0}, Name: {1},
                Address: {2}",
                cust.CustomerId, cust.Name, cust.Address);
            foreach (var cst in cust.Orders){
                Console.WriteLine("Order ID: {0}, Cost: {1}",
                    cst.OrderId,
                    cst.Cost);
            }
        }
    }
}
```

Querying Data by Using LINQ Query Expressions (2-3)

Forming Projections

- When using LINQ queries, the programmer might only require to retrieve specific properties of an entity from the data store; for example only the **Name** property of the **Customer** entity instances. The programmer can achieve this by forming projections in the `select` clause.

- ▶ In addition to simple data retrieval, programmers can use LINQ to perform various other operations, such as forming projections, filtering data, and sorting data.

Snippet

```
public static void DisplayCustomerNames() {  
    using (Model1Container dbContext = new Model1Container()) {  
        IQueryable<String> query = from c in dbContext.Customers select c.Name;  
        Console.WriteLine("Customer Names:");  
        foreach (String custName in query) {  
            Console.WriteLine(custName);  
        }  
    }  
}
```

- ▶ Following code shows a LINQ query that retrieves only the customer names of the Customer entity instances:

Output

```
Customer Names:  
Alex Parker  
Peter Milne
```

Querying Data by Using LINQ Query Expressions (3-3)

Filtering Data

- The `where` clause in a LINQ query enables filtering data based on a Boolean condition, known as the predicate. The `where` clause applies the predicate to the range variable that represents the source elements and returns only those elements for which the predicate is true.

Following code uses the `where` clause to filter customer records:

Snippet

```
public static void DisplayCustomerByName() {  
    using (Model1Container dbContext = new Model1Container()) {  
        IQueryable<Customer> query = from c in dbContext.Customers  
        where c.Name == "Alex Parker" select c;  
        Console.WriteLine("Customer Information:");  
        foreach (Customer cust in query)  
            Console.WriteLine("Customer ID: {0}, Name: {1}, Address: {2}", cust.CustomerId,  
                                cust.Name, cust.Address);  
    }  
}
```

Output

```
Customer Information:  
Customer ID: 1, Name: Alex Parker, Address: 10th Park Street,  
Leo Mount
```

Querying Data by Using LINQ Method-Based Queries

- ▶ Another way to create LINQ queries is by using method-based queries where programmers can directly make method calls to the standard query operator, passing lambda expressions as the parameters.

Snippet

```
public static void DisplayPropertiesMethodBasedQuery() {  
    using (Model1Container dbContext = new Model1Container()) {  
        var query = dbContext.Customers.Select(c => new {  
            CustomerName = c.Name,  
            CustomerAddress = c.Address  
        });  
        Console.WriteLine("Customer Names and Addresses:");  
        foreach (var custInfo in query) {  
            Console.WriteLine("Name: {0}, Address: {1}",  
                custInfo.CustomerName, custInfo.CustomerAddress);  
        }  
    }  
}
```

Output

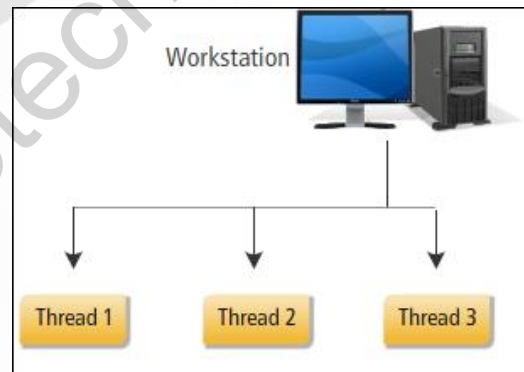
Names and Addresses:

Customer Name: Alex Parker, Address: 10th Park Street, Leo Mount

Name: Peter Milne, Address: Lake View Street, Cheros Mount

Multithreading and Asynchronous Programming

- ▶ In the context of programming language, a thread is a flow of control within an executing application.
- ▶ An application will have at least one thread known as the main thread that executes the application.
- ▶ A programmer can create multiple threads spawning the main thread to concurrently process tasks of the application.
- ▶ A programmer can use various classes and interfaces in the `System.Threading` namespace that provides built-in support for multithreaded programming in the .NET Framework.



Thread Class

- ▶ The `ThreadStart` delegate represents the method that the new thread will execute.
- ▶ Once a thread is instantiated, it can be started by making a call to the `Start()` method of the `Thread` class.
- ▶ Following code instantiates and starts a new thread:

Snippet

```
ServiceReference1.ProductServiceClient client = new
class ThreadDemo {
public static void Print() {
    while (true)
        Console.WriteLine("1");
}
static void Main (string [] args) {
    Thread newThread = new Thread(new ThreadStart(Print));
    newThread.Start();
    while (true)
        Console.WriteLine("2");
}
}
```

Output

The ThreadPool Class

- ▶ The `System.Threading` namespace provides the `ThreadPool` class to create and share multiple threads as and when required by an application.
- ▶ The `ThreadPool` class represents a thread pool, which is a collection of threads in an application.
- ▶ Based on the request from the application, the thread pool assigns a thread to perform a task.
- ▶ When the thread completes execution, it is put back in the thread pool to be reused for another request.
- ▶ The `ThreadPool` class contains a `QueueUserWorkItem()` method that a programmer can call to execute a method in a thread from the thread pool.
- ▶ This method accepts a `WaitCallback` delegate that accepts `Object` as its parameter.
- ▶ The `WaitCallback` delegate represents the method that must execute in a separate thread of the thread pool.

Synchronizing Threads (1-2)

- ▶ When multiple threads have to share data, their activities must be coordinated.
- ▶ This ensures that one thread does not change the data used by the other thread to avoid unpredictable results.
- ▶ For example, consider two threads in a C# program. One thread reads a customer record from a file and the other tries to update the customer record at the same time.
- ▶ In this scenario, the thread that is reading the customer record might not get the updated value as the other thread might be updating the record at that instance.
- ▶ To avoid such situations, C# allows programmers to coordinate and manage the actions of multiple threads at a given time using the following thread synchronization mechanisms.
 - ◆ **Locking using the lock Keyword**
 - Locking is the process that gives control to execute a block of code to one thread at one point of time.
 - The block of code that locking protects is referred to as a critical section. Locking can be implemented using the lock keyword. When using the lock keyword, the programmer must pass an object reference that a thread must acquire to execute the critical section.
 - For example, to lock a section of code within an instance method, the reference to the current object can be passed to the lock.

Synchronizing Threads (2-2)

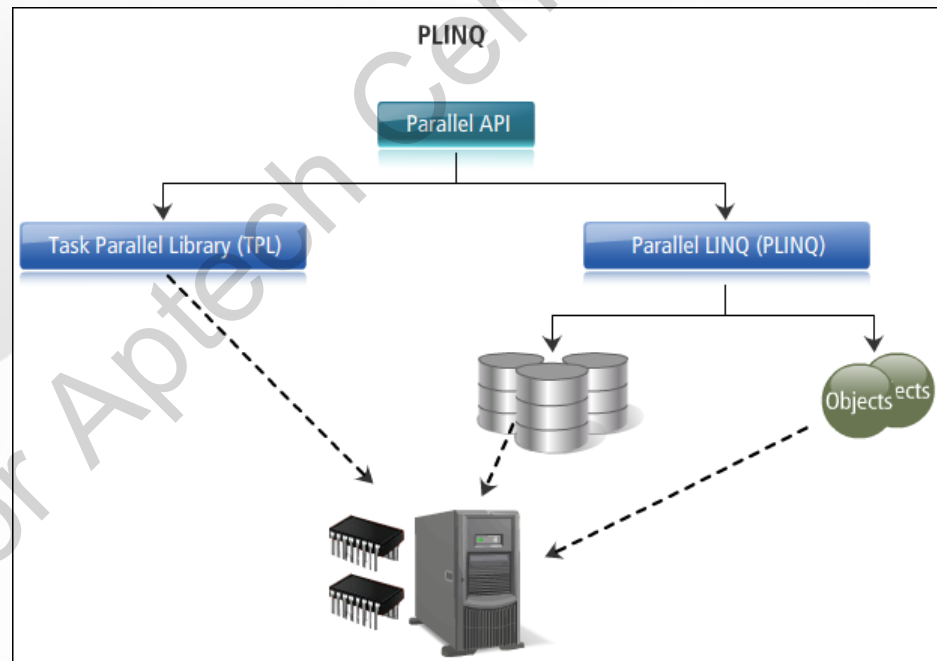
❖ Synchronization Events

- The locking mechanism used for synchronizing threads is useful for protecting critical sections of code from concurrent thread access.
- However, locking does not allow communication between threads. To enable communication between threads while synchronizing them, C# supports synchronization events.
- A synchronization event is an object that has one of two states: signaled and un-signaled.
- When a synchronized event is in un-signaled state, threads can be made suspended until the synchronized event comes to the signaled state.
- The `AutoResetEvent` class of the `System.Threading` namespace represents a synchronization event that changes automatically from signaled to un-signaled state any time a thread becomes active.
- The `AutoResetEvent` class provides the `WaitOne()` method that suspends the current thread from executing until the synchronized event comes to the signaled state.
- The `Set()` method of the `AutoResetEvent` class changes the state of the synchronized event from un-signaled to signaled.

For Aptech Centre Use Only

The Task Parallel Library

- ▶ Modern computers contain multiple CPUs. In order to take advantage of the processing power that computers with multiple CPUs deliver, a C# application must execute tasks in parallel on multiple CPUs. This is known as parallel programming.
- ▶ To make parallel and concurrent programming simpler, the .NET Framework introduced Task Parallel Library (TPL). TPL is a set of public types and APIs in the `System.Threading` and `System.Threading.Tasks` namespaces.



The Task Class

- ▶ This method passes the task to the task scheduler that assigns threads to perform the work.
- ▶ To ensure that a task completes before the main thread exits, a programmer can call the `Wait()` method of the `Task` class the programmer can call the `WaitAll()` method passing an array of the `Tasks` objects that have started.
- ▶ The `Task` class also provides a `Run()` method to create and start a task in a single operation.

Snippet

```
class TaskDemo {  
    private static void printMessage() {  
        Console.WriteLine("Executed by a Task");  
    }  
  
    static void Main (string [] args) {  
        Task task1 = new Task(new Action(printMessage));  
        task1.Start();  
        Task task2 = Task.Run(() => printMessage());  
        task1.Wait();  
        task2.Wait();  
        Console.WriteLine("Exiting main method");  
    }  
}
```

Executed by a Task
Executed by a Task
Exiting main method

Output

Obtaining Results from a Task

- ▶ Often a C# program would require some results after a task completes its operation.
- ▶ To provide results of an asynchronous operation, the .NET Framework provides the `Task<T>` class that derives from the `Task` class.
- ▶ In the `Task<T>` class, `T` is the data type of the result that will be produced.
- ▶ To access the result, call the `Result` property of the `Task<T>` class.

For Aptech Centre Users Only

Task Continuation

- ▶ When multiple tasks execute in parallel, it is common for one task, known as the antecedent to complete an operation and then invoke a second task, known as the continuation task.
- ▶ Such task continuation can be achieved by calling a `ContinueWith()` overloaded methods of the antecedent task.
- ▶ The simplest form of the `ContinueWith()` method accepts a single parameter that represents the task to be executed once the antecedent completes.
- ▶ The `ContinueWith()` method returns the new task.
A programmer can call the `Wait()` method on the new task to wait for it to complete.



Task Cancellation

- ▶ TPL provides the `CancellationTokenSource` class in the `System.Threading` namespace that can be used to cancel a long running task.
- ▶ The `CancellationTokenSource` class has a `Token` property that returns an object of the `CancellationToken` struct.
- ▶ This object propagates notification that a task should be canceled. While creating a task that can be cancelled, the `CancellationToken` object must be passed to the task.
- ▶ The `CancellationToken` struct provides an `IsCancellationRequested` property that returns true if a cancellation has been requested.
- ▶ A long running task can query the `IsCancellationRequested` property to check whether a cancellation request is being made, and if so elegantly end the operation.
- ▶ A cancellation request can be made by calling the `Cancel()` method of the `CancellationTokenSource` class.
- ▶ While cancelling a task, a programmer can call the `Register()` method of the `CancellationToken` struct to register a callback method that receives a notification when the task is cancelled.

Parallel Loops (1-2)

- ▶ TPL introduces a `Parallel` class in the `System.Threading.Tasks` namespace which provides methods to perform parallel computation of loops, such as `for` loops and `foreach` loops.
- ▶ The `For()` method is a static method in the `Parallel` class that enables executing a `for` loop with parallel iterations.
- ▶ As the iterations of a loop done using the `For()` method are parallel, the order of iterations might vary each time the `For()` method executes.
- ▶ The `For()` method has several overloaded versions.
- ▶ The most commonly used overloaded `For()` method accepts the following three parameters in the specified order:
 - ▶ An `int` value representing the start index of the loop.
 - ▶ An `int` value representing the end index of the loop.
 - ▶ A `System.Action<Int32>` delegate that is invoked once per iteration.

Snippet

```
static void Main (string [] args) {
    Console.WriteLine("\nUsing traditional for loop");
    for (int i = 0; i <= 10; i++) {
        Console.WriteLine("i = {0} executed by thread with ID {1}", i,
            Thread.CurrentThread.ManagedThreadId);
        Thread.Sleep(100);
    }
    Console.WriteLine("\nUsing Parallel For");
    Parallel.For(0, 10, i => {
        Console.WriteLine("i = {0} executed by thread with ID {1}", i,
            Thread.CurrentThread.ManagedThreadId);
        Thread.Sleep(100);
    });
}
```

Parallel LINQ (PLINQ)

- ▶ LINQ to Object refers to the use of LINQ queries with enumerable collections, such as `List<T>` or arrays.
- ▶ PLINQ is the parallel implementation of LINQ to Object. While LINQ to Object sequentially accesses an in-memory `IEnumerable` or `IEnumerable<T>` data source, PLINQ attempts parallel access to the data source based on the number of processor in the host computer.

Snippet

```
string[] arr = new string[] { "Peter", "Sam",  
    "Philip", "Andy", "Philip", "Mary", "John", "Pamela"};  
var query = from string name in arr select name;  
Console.WriteLine("Names retrieved using sequential LINQ");  
foreach (var n in query)  
{  
    Console.WriteLine(n);  
}  
  
var plinqQuery = from string name in arr.AsParallel()  
    select name;  
Console.WriteLine("Names retrieved using PLINQ");  
foreach (var n in plinqQuery)  
{  
    Console.WriteLine(n);  
}
```

Concurrent Collections (1-2)

- ▶ The collection classes of the `System.Collections.Generic` namespace provides improved type safety and performance compared to the collection classes of the `System.Collections` namespace.
- ▶ However, the collection classes of the `System.Collections.Generic` namespace are not thread safe.
- ▶ As a result, programmer must provide thread synchronization code to ensure integrity of the data stored in the collections.
- ▶ To address thread safety issues in collections, the .NET Framework provides concurrent collection classes in the `System.Collections.Concurrent` namespace.
- ▶ These classes being thread safe relieves programmers from providing thread synchronization code when multiple threads simultaneously accesses these collections.
- ▶ Important classes of the `System.Collections.Concurrent` namespace are as follows:

`ConcurrentDictionary<T
Key, TValue>`

- Is a thread-safe implementation of a dictionary of key-value pairs.

`ConcurrentQueue<T>`

- Is a thread-safe implementation of a queue.

`ConcurrentStack<T>`

- Is a thread-safe implementation of a stack.

`ConcurrentBag<T>`

- Is a thread-safe implementation of an unordered collection of elements.

Concurrent Collections (2-2)

Following code uses multiple threads to add elements to an object of `ConcurrentDictionary<string, int>` class:

Snippet

```
class CollectionDemo
{
    static ConcurrentDictionary<string, int> dictionary = new
    ConcurrentDictionary<string, int>();
    static void AddToDictionary()
    {
        for (int i = 0; i < 100; i++)
        {
            dictionary.TryAdd(i.ToString(), i);
        }
    }

    static void Main(string[] args)
    {
        Thread thread1 = new Thread(new ThreadStart(AddToDictionary));
        Thread thread2 = new Thread(new ThreadStart(AddToDictionary));
        thread1.Start();
        thread2.Start();
        thread1.Join();
        thread2.Join();
        Console.WriteLine("Total elements in dictionary: {0}",
        dictionary.Count());
    }
}
```

Asynchronous Methods

- ▶ TPL provides support for asynchronous programming through two new keywords: `async` and `await`.
- ▶ A method marked with the `async` keyword notifies the compiler that the method will contain at least one `await` keyword. The `await` keyword is applied to an operation to temporarily stop the execution of the `async` method until the operation completes.
- ▶ In the meantime, control returns to the `async` method's caller. Once the operation marked with `await` completes, execution resumes in the `async` method.
- ▶ A method marked with the `async` keyword can have one of the following return types: `void`, `Task`, and `Task<TResult>`

Snippet

```
class AsyncAwaitDemo {
    static async void PerformComputationAsync() {
        Console.WriteLine("Entering asynchronous method");
        int result = await new ComplexTask().AnalyzeData();
        Console.WriteLine(result.ToString());
    }
    static void Main(string[] args) {
        PerformComputationAsync();
        Console.WriteLine("Main thread executing.");
        Console.ReadLine();
    }
}

class ComplexTask {
    public Task<int> AnalyzeData() {
        Task<int> task = new Task<int>(GetResult);
        task.Start();
        return task;
    }
    public int GetResult() {
        /*Pause Thread to simulate time consuming operation*/
        Thread.Sleep(2000);
        return new Random().Next(1, 1000);
    }
}
```

Dynamic Programming

- ▶ A programmer using a dynamic type is not required to determine the source of the object's value during application development.
- ▶ However, any error that escapes compilation checks causes a run-time exception.
- ▶ To understand how dynamic types bypasses compile type checking, consider the following code:

Snippet

```
class DemoClass
{
    public void Operation(String name)
    {
        Console.WriteLine("Hello {0}", name);
    }
}
class DynamicDemo
{
    static void Main(string[] args)
    {
        dynamic dynaObj = new DemoClass();
        dynaObj.Operation();
    }
}
```

Summary

- ▶ Anonymous methods allow you to pass a block of unnamed code as a parameter to a delegate.
- ▶ Extension methods allow you to extend different types with additional static methods.
- ▶ You can create an instance of a class without having to write code for the class beforehand by using a new feature called anonymous types.
- ▶ Partial types allow you to split the definitions of classes, structs, and interfaces to store them in different C# files.
- ▶ You can define partial types using the partial keyword.
- ▶ Nullable types allow you to assign null values to the value types.
- ▶ Nullable types provide two public read-only properties, HasValue and Value.
- ▶ ADO.NET is a database technology of .NET Framework that is used to connect an application system and a database server.
- ▶ Entity Framework refers to an open-source ORM framework intended for .NET applications supported by Microsoft.
- ▶ There are many features of Entity Framework 6 that have not been added in Entity Framework Core.
- ▶ EF Core 5.0 cannot be used with .NET Framework applications.
- ▶ For new application development, it is recommended to use EF Core on .NET 5.0.