Session: 15

Advanced Concepts of C#

Objectives

- Describe system-defined generic delegates
- Define lambda expressions
- Explain query expressions
- Describe Windows Communication Framework (WCF)
- Explain parallel programming
- Explain dynamic programming

System-Defined Generic Delegates 1-3

- A delegate is a reference to a method. Consider an example to understand this.

Example

- You create a delegate **`CalculateValue`** to point to a method that takes a `string` parameter and returns an `int`.
- You need not specify the method name at the time of creating the delegate.
- At some later stage in your code, you can instantiate the delegate by assigning it a method name.
- The .NET Framework and C# have a set of predefined generic delegates that take a number of parameters of specific types and return values of another type.
- The advantage of these predefined generic delegates is that they are ready for reuse with minimal coding.

System-Defined Generic Delegates 2-3

- Following are the commonly used predefined generic delegates:

`Func<TResult>() Delegate`	`Func<T, TResult>(T arg) Delegate`	`Func<T1, T2, TResult>(T1 arg1, T2 arg2) Delegate`
• It represents a method having zero parameters and returns a value of type **`TResult`**.	• It represents a method having one parameter of type **T** and returns a value of type **`TResult`**.	• It represents a method having two parameters of type **T1** and **T2** respectively and returns a value of type **`TResult`**.

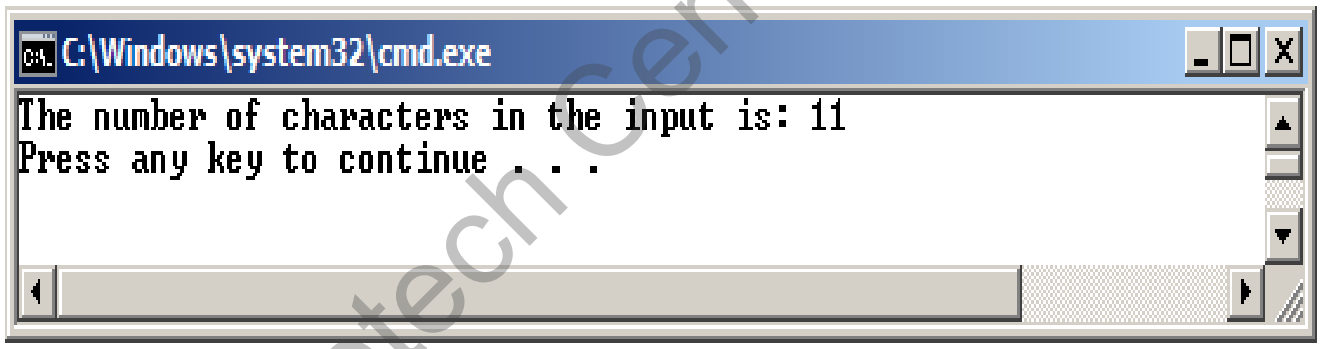
- The following code determines the length of a given word or phrase:

Syntax

```
/// <summary>
/// Class WordLength determines the length of a given word or phrase
/// </summary>
public class WordLength{
     public static void Main() {
          // Instantiate delegate to reference Count method
          Func<string, int> count = Count;
          string location = "Netherlands";
          // Use delegate instance to call Count method
          Console.WriteLine("The number of characters in the input is:
          {0} ",count(location).ToString());
     }
     private static int Count(string inputString) {
          return inputString.Length;
     }
}
```

- The code:
  - Makes use of the `Func<T, TResult> (T arg)` predefined generic delegate that takes one parameter and returns a result.

- The following figure shows the use of a predefined generic delegate:



Lambda Expressions 1-3

- 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.
- A lambda expression:
  - Is an anonymous expression that can contain expressions and statements and enables to simplify development through inline coding.
  - In simple terms, a lambda expression is an inline expression or statement block having a compact syntax and can be used wherever a delegate or anonymous method is expected.

- The following syntax shows a lambda expression that can be used wherever a delegate or anonymous method is expected:

Syntax

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

- where,
  - **`parameter-list`**: is an explicitly typed or implicitly typed parameter list
  - =>: is the lambda operator
  - **`expression or statements`**: are either an expression or one or more statements

Example

- The following code is a lambda expression:

```
word => word.Length;
```

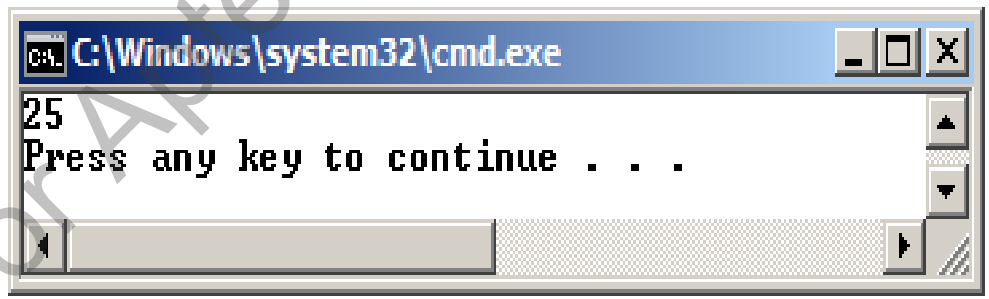
- Consider a complete example to illustrate the use of lambda expressions.
- Assume that you want to calculate the square of an integer number.
- You can use a method **`Square()`** and pass the method name as a parameter to the `Console.WriteLine()` method.

- The 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));
      }
}
```

- In the code:
  - The => operator is pronounced as 'goes to' or 'go to' in case of multiple parameters.
  - Here, the expression, **`input =>input * input`** means, given the value of input, calculate input multiplied by **`input`** and return the result.
  - The following figure shows an example that returns the square of an integer number:



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
```

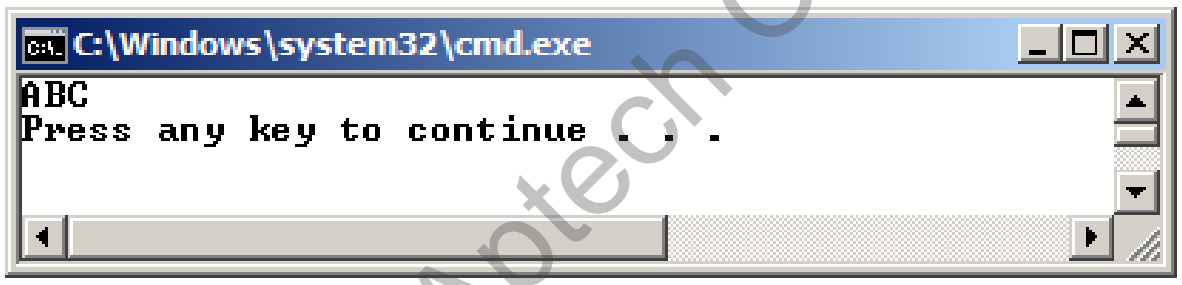
- To use a lambda expression:
  - Declare a delegate type which is compatible with the lambda expression.
  - Then, create an instance of the delegate and assign the lambda expression to it. After this, you will invoke the delegate instance with parameters, if any.
  - This will result in the lambda 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"));
      }
}
```

- 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



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.

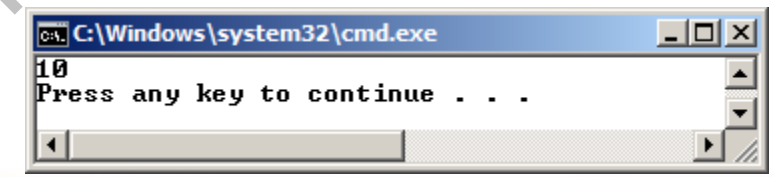
```
(string str, string str1)=> { return (str==str1);}
```

- 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");
}
}
```

Output



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

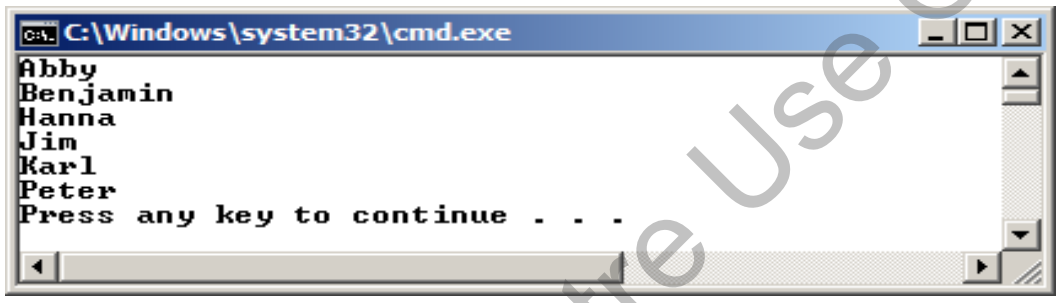
- The following shows how to use the `OrderBy` operator with the lambda operator to sort a list of names:

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:



Query Expressions 1-3

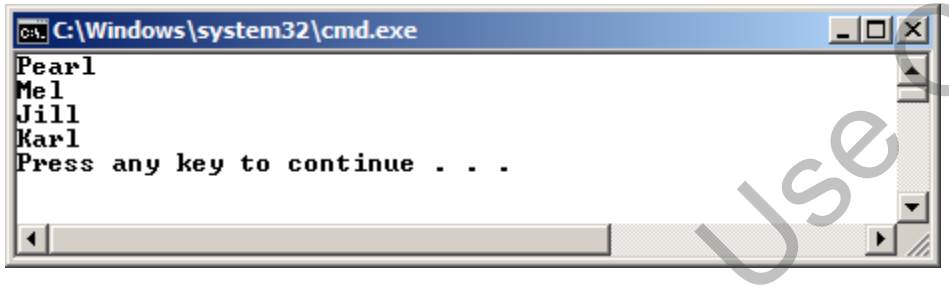
- A query expression is a query that is written in query syntax using clauses such as `from`, `select`, and so forth. These clauses are an inherent part of a LINQ query.
- LINQ is a set of technologies introduced in Visual Studio 2008 that simplifies working with data present in various formats in different data sources. LINQ provides a consistent model to work with such data.
- Developers can create and use query expressions, which are used to query and transform data from a data source supported by LINQ.
- A `from` clause must be used to start a query expression and a `select` or `group` clause must be used to end the query expression.
- The following code shows a simple example of a query expression. Here, a collection of strings representing names is created and then, a query expression is constructed to retrieve only those names that end with 'l':

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);
    }
}
```

- The following displays the query expression example:

Output



- Whenever a compiler encounters a query expression, internally it converts it into a method call, using extension methods.
- So, an expression such as the one shown in code is converted appropriately.

```
IEnumerable<string> words = from word in names
where word.EndsWith("l")
select word;
```

- **After conversion:**

```
IEnumerable<string> words = names.Where(word
=>word.EndsWith("l"));
```

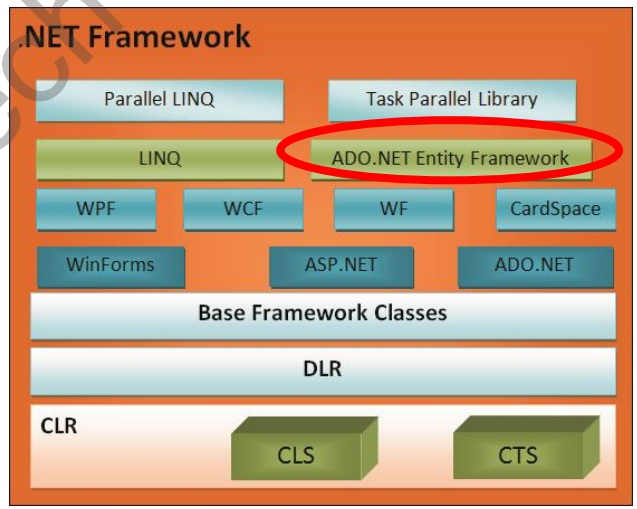
- Though this line is more compact, the SQL way of writing the query expression is more readable and easier to understand.

- 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 persist and retrieve data, an application first needs to connect with the data store.
- The application must also ensure that the definitions and relationships of classes or objects are mapped with the database tables and table relationships.
- Finally, the application needs to provide the data access code to persist and retrieve data.
- These operations can be achieved using ADO.NET, which is a set of libraries that allows an application to interact with data sources.
- 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.



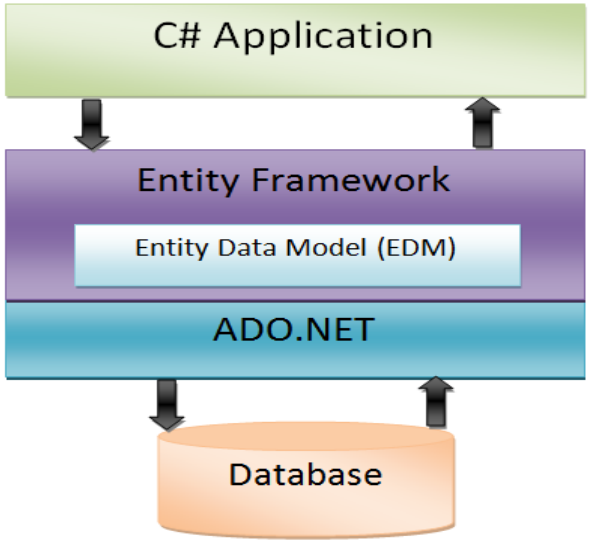
The Entity Data Model

- 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.

- The figure shows the role of EDM in the Entity Framework architecture:



- Entity Framework eliminates the need to write most of the data-access code that otherwise need to 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.

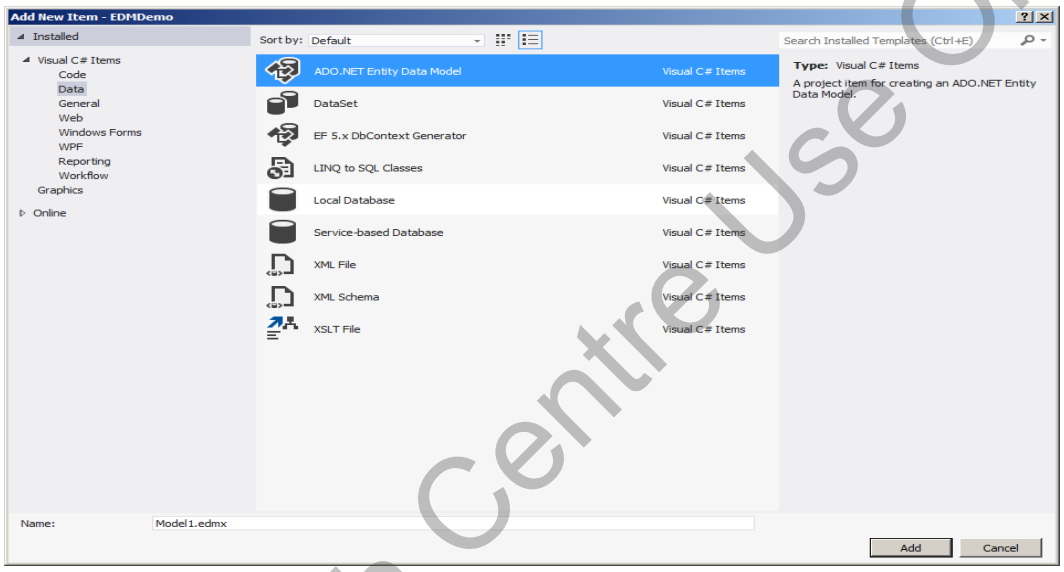
The code-first approach

- 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 1-4

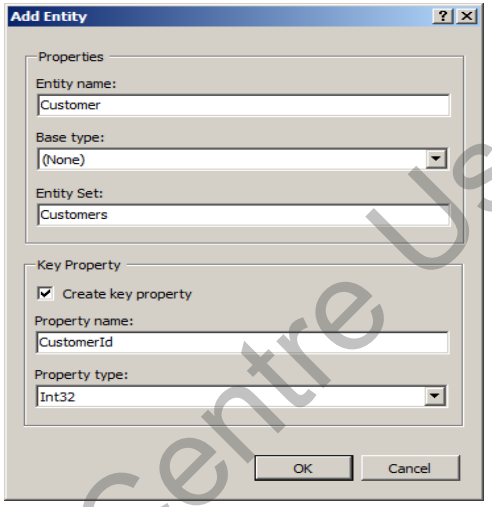
- Visual Studio 2012 provides support for creating and using EDM in C# application.
- Programmers can use the Entity Data Model wizard to create a model in an application.
- After creating a model, programmer can add entities to the model and define their relationship using the Entity Framework designer.
- The information of the model is stored in an `.edmx` file.
- Based on the model, programmer can use Visual Studio 2012 to automatically generate the database objects corresponding to the entities.
- Finally, the programmer can use LINQ queries against the entities to retrieve and update data in the underlying database.
- To create an entity data model and generate the database object, a programmer needs to 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:

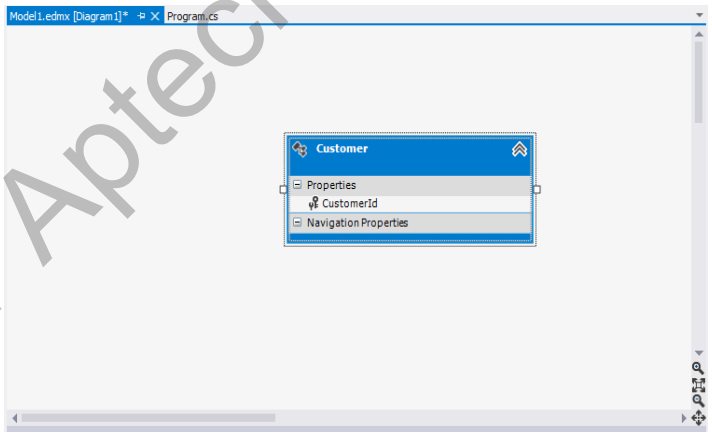


- Click **Add**. The **Entity Data Model Wizard** appears.
- Select **Empty model**.
- Click **Finish**. The **Entity Data Model Designer** is displayed.
- Right-click the Entity Data Model Designer, and select **Add New → Entity**. The **Add Entity** dialog box is displayed.

- Enter **Customer** in the **Entity name** field and **CustomerId** in the **Property name** field, as shown in the following figure:



- Click **OK**. The Entity Data Model Designer displays the new **Customer** entity, as shown in the following figure:

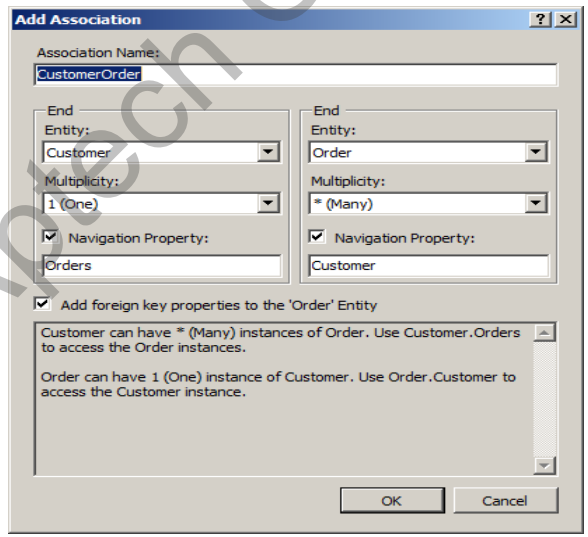


Creating an Entity Data Model 4-4

- 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.

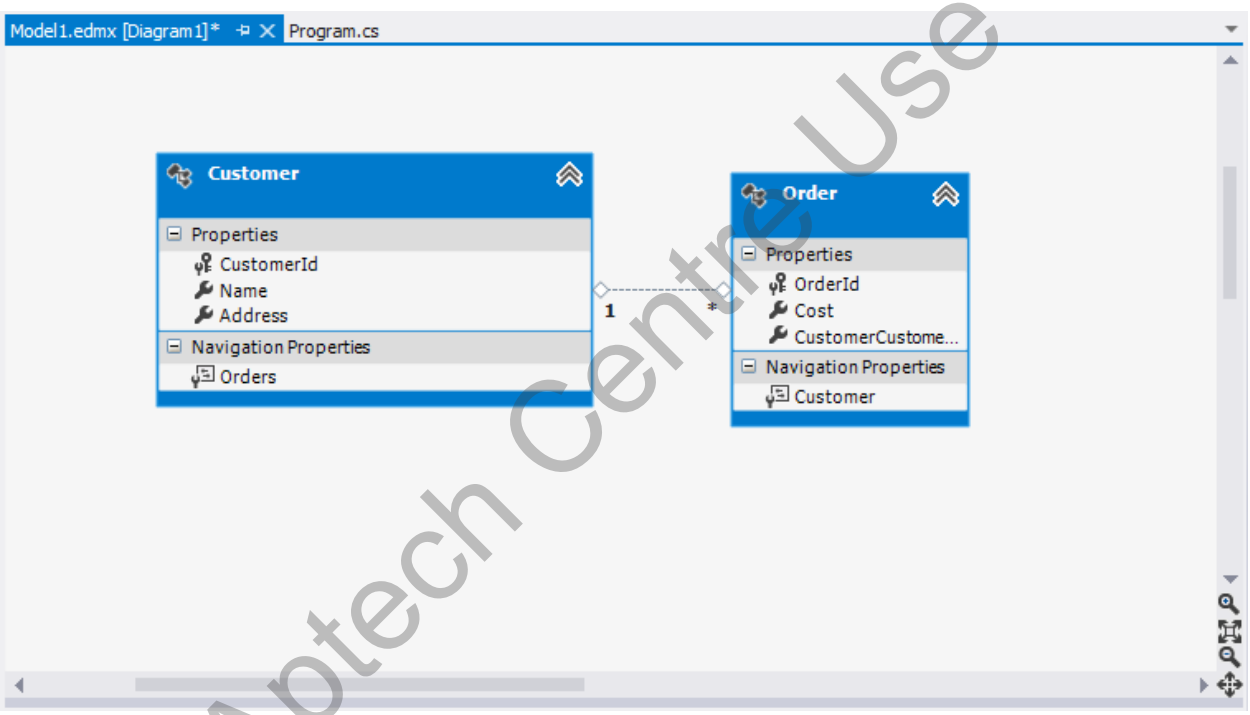
Defining Relationships 1-2

- 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-hand **End** section point to **Post** with a multiplicity of *(**Many**). Accept the default setting for the other fields, as shown in the following figure:



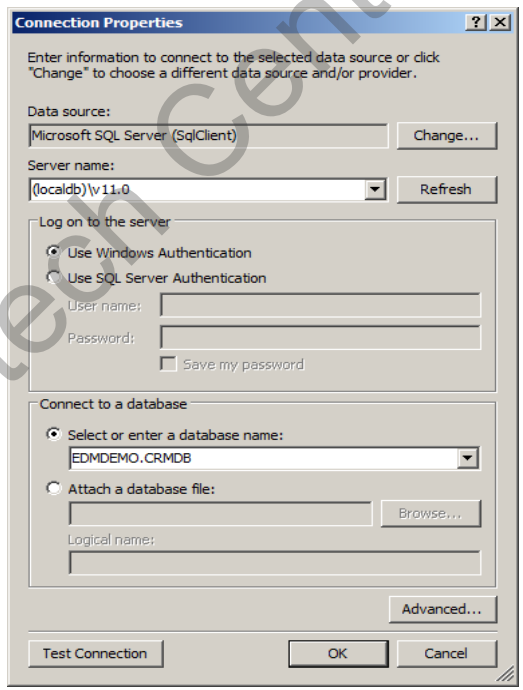
Defining Relationships 2-2

- Click **OK**. The Entity Data Model Designer displays the entities with the defined relationship, as shown in the following figure:



Creating Database Objects 1-2

- After designing the model of the application, the programmer needs to 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 as shown in the following figure:

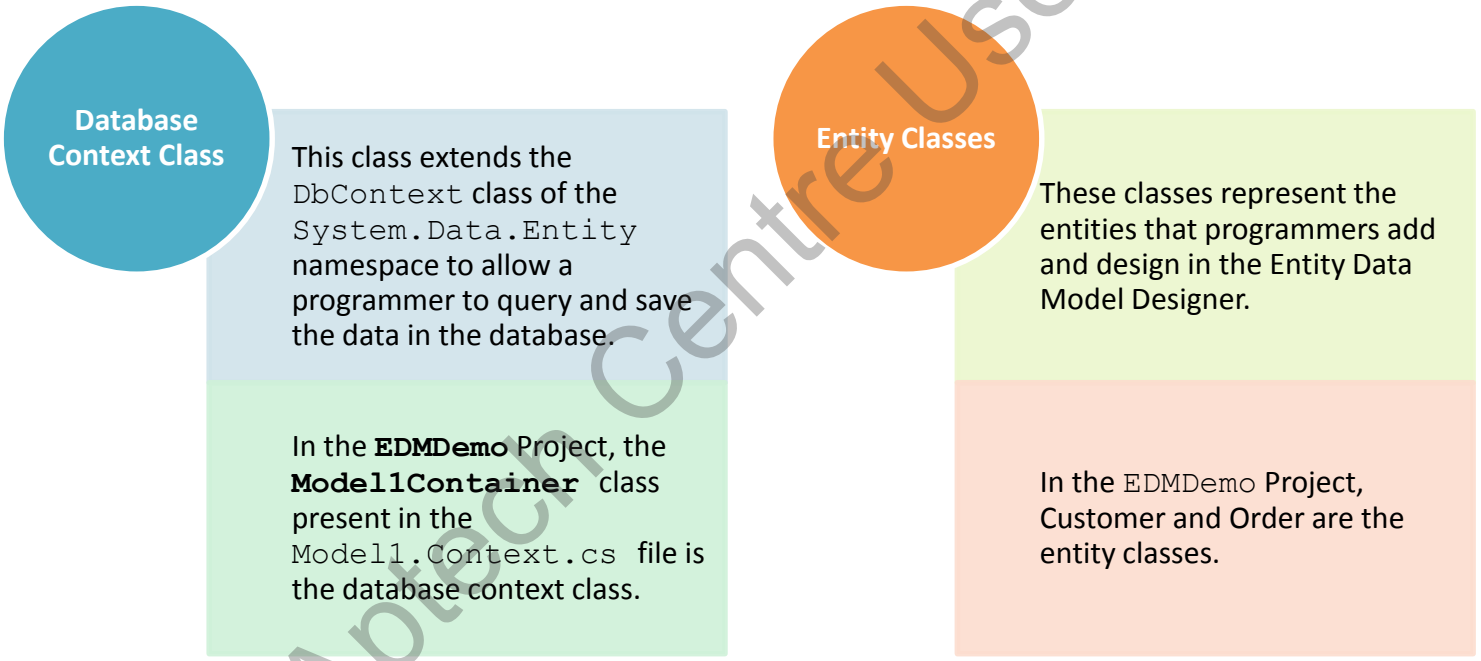


Creating Database Objects 2-2

- 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 1-3

- 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:



- The following code shows the `Main()` method that creates an persists `Customer` and `Order` entities:

Snippet

```
class Program{

  static void Main(string[] args)    {
    using (Model1Container dbContext = new Model1Container()){
     Console.Write("Enter Customer name: ");
     var name = Console.ReadLine();
     Console.Write("Enter Customer Address: ");
     var address = Console.ReadLine();
     Console.Write("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.");
      }

    }
}
```

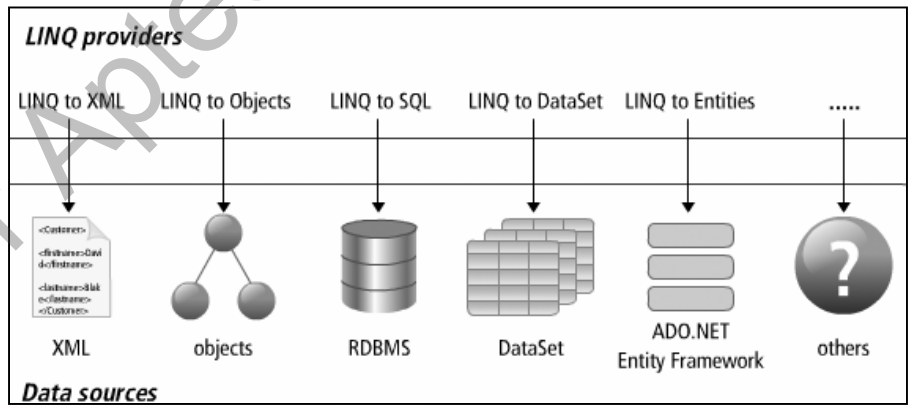
Output

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

- The code:
  - Prompts and accept customer and order information from the console.
  - Then, the **Customer** and **Order** objects are created and initialized with data. The Order object is added to the **Orders** property of the **Customer** object.
  - The **Orders** property which is of type, **ICollection<Order>** enables adding multiple **Order** objects to a **Customer** object, based on the one-to-many relationship that exists between the **Customer** and **Order** entities.
  - Then, the database context object of type **Model1Container** is used to add the **Customer** object to the database context.
  - Finally, the call to the **SaveChanges()** method persists the **Customer** object to the database.

Querying Data by Using LINQ Query Expressions 1-4

- LINQ provides a consistent programming model to create standard query expression syntax to query different types of data sources.
- However, different data sources accept queries in different formats. To solve this problem, LINQ provides the various LINQ providers, such as LINQ to Entities, LINQ to SQL, LINQ to Objects, and LINQ to XML.
- 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 needs 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.



Querying Data by Using LINQ Query Expressions 2-4

- The following code creates and executes a query to retrieve the records of all `Customer` entities along with the associated `Order` entities:

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);
          }
       }
   }
}
```

- In the code:
  - The `from` clause specifies the data source from where the data has to be retrieved.
  - `dbContext` is an instance of the data context class that provides access to the `Customers` data source, and `c` is the range variable.
  - When the query is executed, the range variable acts as a reference to each successive element in the data source.
  - The `select` clause in the LINQ query specifies the type of the returned elements as an `IQueryable<Customer>` object.
  - The `foreach` loops iterate through the results of the query returned as an `IQueryable<Customer>` object to print the customer and order details.

Querying Data by Using LINQ Query Expressions 3-4

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

Forming Projections

- When using LINQ queries, the programmer might only need 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.

- The following code shows a LINQ query that retrieves only the customer names of the `Customer` entity instances:

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);
        }
   }
}
```

- In the code:
  - The `select` method retrieves a sequence of customer names as an `IQueryable<String>` object and the `foreach` loop iterates through the result to print out the names.

Output

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

Querying Data by Using LINQ Query Expressions 4-4

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.

The 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);
          }
   }
}
```

- The code:
  - Uses the `where` clause to retrieve information of the customer with the name `Alex Parker`. The `foreach` statement iterate through the result to print the information of the customer.

Output

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

Querying Data by Using LINQ Method-Based Queries 1-2

- The LINQ queries used so far are created using query expression syntax.
- Such queries are compiled into method calls to the standard query operators, such as `select`, `where`, and `orderby`.
- 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.
- The following code uses the `Select` method to project the **`Name`** and **`Address`** properties of **`Customer`** entity instances into a sequence of anonymous types:

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);

       }
     }
    }
```

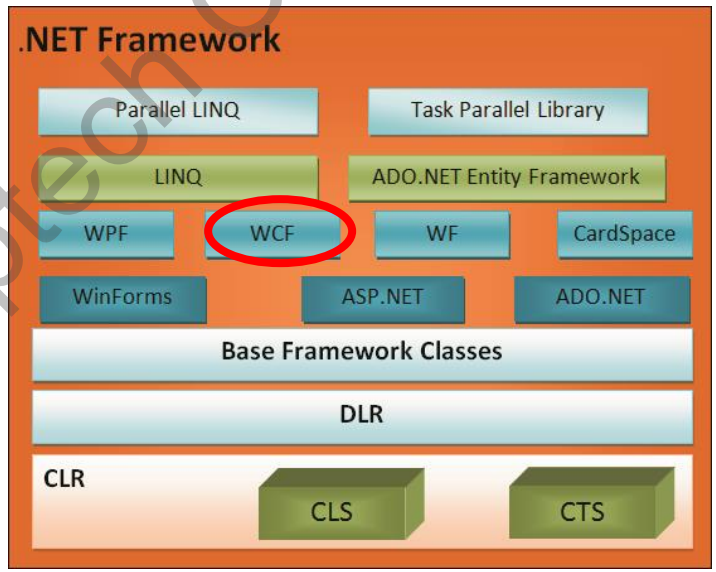
Querying Data by Using LINQ Method-Based Queries 2-2

Output

```
Customer Names and Addresses:
Name: Alex Parker, Address: 10th Park Street, Leo Mount
Name: Peter Milne, Address: Lake View Street, Cheros
Mount
```

- Similarly, you can use the other operators such as `where`, `GroupBy`, `Max`, and so on through method-based queries.

- Windows Communication Foundation (WCF) is a framework for creating loosely-coupled distributed application based on Service Oriented Architecture (SOA).
- SOA is an extension of distributed computing based on the request/response design pattern.
- SOA allows creating interoperable services that can be accessed from heterogeneous systems.
- Interoperability enables a service provider to host a service on any hardware or software platform that can be different from the platform on the consumer end.



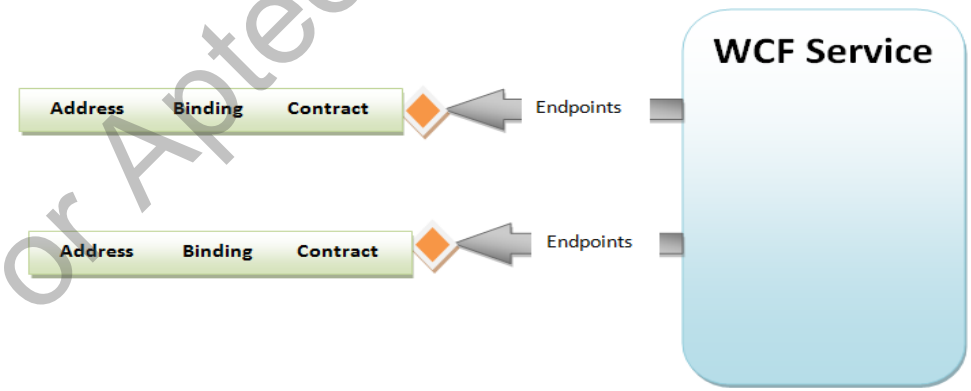
- Microsoft has several service-oriented distributed technologies, such as ASP.NET Web Services, .NET Remoting, Messaging, and Enterprise Services.
- Each of these technologies has their own set of infrastructure, standards, and API to enable solving different distributed computing requirements.
- However, there was a need of an interoperable solution that provides the benefits of the distributed technologies in a simple and consistent manner.
- As a solution, Microsoft introduced WCF services, which is a unified programming model for service-oriented applications. In WCF, a client communicates with a service using messages.
- In a WCF communication, a service defines one or more service endpoints, that defines the information required to exchange messages.
- A service provides the information through service endpoints in the form of metadata.
- Based on the metadata exposed by a service, the client initiates a communication with the service by sending a message to the service endpoint.
- The service on receiving a communication message responds to the message.

Service Endpoints

- In a WCF service, a service endpoint provides the following information, also known as the ABC's of WCF:

Address	• Specifies the location where messages can be sent.
Binding	• Specifies the communication infrastructure that enables communication of messages. Binding is formed by a stack of components implemented as channels. At the minimum, a binding defines the transport mechanism, such as HTTP or TCP and the encoding, such as text or binary that is being used to communicate messages.
Contract	• Defines a set of messages that can be communicated between a client and a service.

- The following figure shows service endpoints in a WCF Service:



- WCF operations are based on standard contracts that a WCF service provides that are as follows:
  - **Service Contract**:
    - Describes what operations a client can perform on a service.
    - The following code shows a service contract applied to an `IProductService` interface:

Snippet

```
[ServiceContract]
public interface IProductService{
}
```

    - The code uses the `ServiceContract` attribute on the `IProductService` interface to specify that the interface will provide one or more operations that client can invoke using WCF.
  - **Operational Contract:**
    - The following code shows two operational contracts applied to the `IProductService` interface:

Snippet

```
ServiceContract]
public interface IProductService
{
 [OperationContract]
 String GetProductName (int productId);

 [OperationContract]
  IEnumerable<ProductInformation> GetProductInfo (int productId);

 }
```

- The code:
  - Uses the `OperationContract` attribute on the `GetProductName()` and `GetProductInfo()` methods to specify that these methods can service WCF clients.
  - In the first operational contract applied to the `GetProductName()` method, no additional data contract is required as both the parameter and return types are primitives.
  - However, the second operational contract returns a complex `ProductInfo` type and therefore, must have a data contract defined for it.

Data Contract and Data Members

Specifies how the WCF infrastructure should serialize complex types for transmitting its data between the client and the service. A data contract can be specified by applying the `DataContract` attribute to the complex type, which can be a class, structure, or enumeration. Once a data contract is specified for a type, the `DataMember` attribute must be applied to each member of the type.

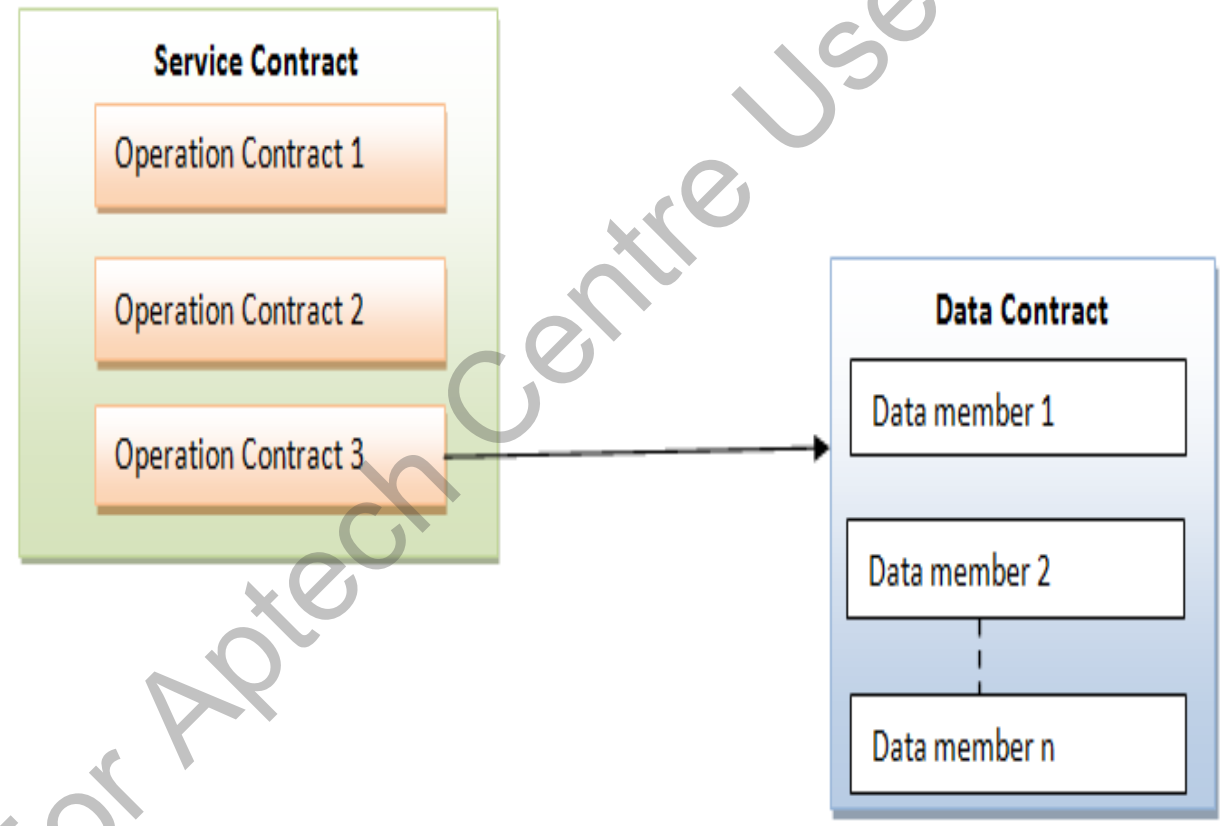
- The following code shows a data contract applied to the `ProductInformation` class:

Snippet

```
[DataContract]
public class ProductInformation
{

  [DataMember]
  public int ProductId
  {
    get; set;
  }
  [DataMember]
  public String ProductName
  {
    get; set;
  }

  [DataMember]
  public int ProductPrice
  {
    get; set;
  }
}
```

- The code:
  - Applies the `DataContract` attribute to the `ProductInformation` class and the `DataMember` attribute to the `ProductId`, `ProductName`, and `ProductPrice` attributes.

- The following figure shows the relationships between the different contracts in a WCF application:



Creating the Implementation Class 1-2

- After defining the various contracts in a WCF application, the next step is to create the implementation class. This class implements the interface marked with the `ServiceContract` attribute.
- The following code shows the `ProductService` class that implements the `IProductService` interface:

Snippet

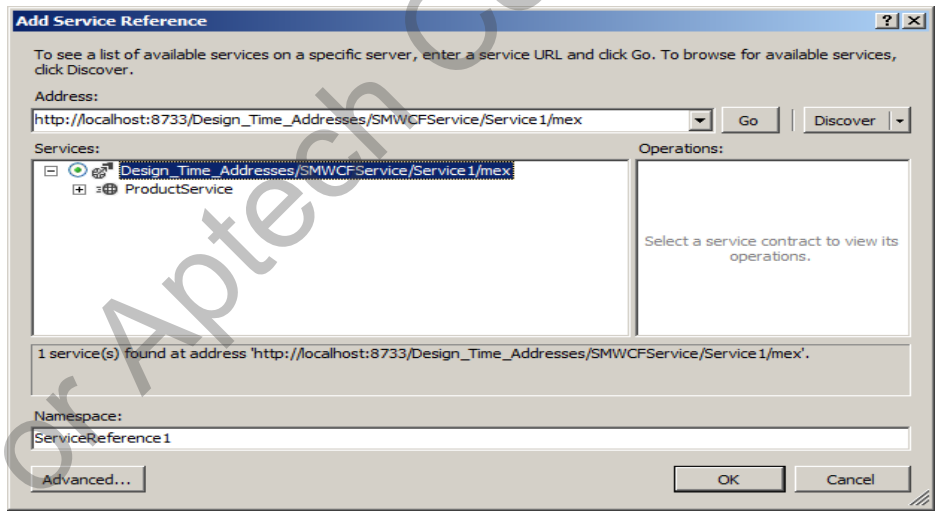
```
public class ProductService : IProductService{
List<ProductInformation> products = new
List<ProductInformation>() ;
    public ProductService()    {
      products.Add(new ProductInformation{ ProductId = 001,
       ProductName = "Hard Drive",ProductPrice= 175 });
      products.Add(new ProductInformation { ProductId = 002,
       ProductName = "Keyboard", ProductPrice = 15 });
      products.Add(new ProductInformation { ProductId = 003,
       ProductName = "Mouse",ProductPrice = 15 });
    }


    public string GetProductName(int productId)    {
      IEnumerable<string> Product
             = from product in products
              where product.ProductId == productId
              select product.ProductName;
      return Product.FirstOrDefault();
    }


    public IEnumerable<ProductInformation> GetProductInfo(int productId){

      IEnumerable<ProductInformation> Product =
        from product in products
        where product.ProductId == productId
        select product ;
      return Product;
    }
}
```

Creating the Implementation Class 2-2

- The code:
  - Creates the **`ProductService`** class that implements the **`IProductService`** interface marked as the service contract.
  - The constructor of the **`ProductService`** class initializes a `List` object with **`ProductInformation`** objects.
  - The **`ProductService`** class implements the **`GetProductName()`** and **`GetProductInfo()`** methods marked as operational contract in the **`IProductService`** interface.
  - The **`GetProductName()`** method accepts a product ID and performs a LINQ query on the `List` object to retrieve the name and price of the **`ProductInformation`** object.
  - The **`GetProductInfo()`** method accepts a product ID and performs a LINQ query on the `List` object to retrieve a **`ProductInfo`** object in an **`IEnumerable`** object.

Creating a Client to Access a WCF Service 1-2

- After creating a WCF service, a client application can access the service using a proxy instance.
- To use the proxy instance and access the service, the client application requires a reference to the service.
- In the Visual Studio 2012 IDE, you need to perform the following steps to add a service reference to a client project:
  - In the **`Solution Explorer`**, right-click the **`Service References`** node under the project node and select **`Add References`**. The **`Add Service Reference`** dialog box is displayed.
  - Click **Discover**. The **`Add Service Reference`** dialog box displays the hosted WCF service, as shown in the following figure:



  - Click **OK**. A reference to the WCF service is added to the project.

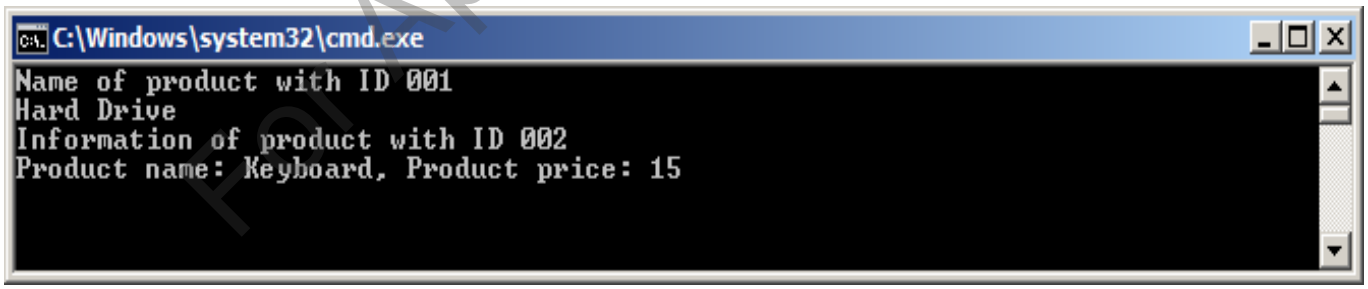
Creating a Client to Access a WCF Service 2-2

- The following code shows the `Main()` method of a client that accesses a WCF service using the proxy instance:

Snippet

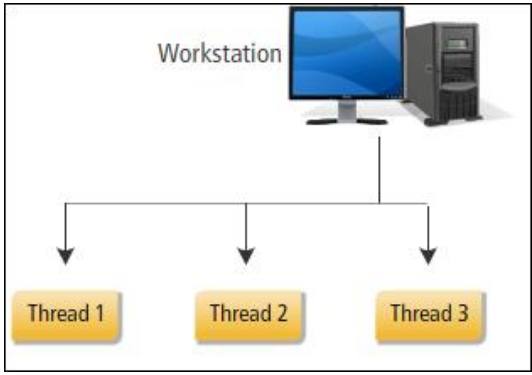
```
 ServiceReference1.ProductServiceClient client = new
     ServiceReference1.ProductServiceClient();
Console.WriteLine("Name of product with ID 001");
Console.WriteLine(client.GetProductName(001));

Console.WriteLine("Information of product with ID 002");
ServiceReference1.ProductInformation[] productArr=client.GetProductInfo(002);
  foreach(ServiceReference1.ProductInformation product in productArr){
        Console.WriteLine("Product name: {0}, Product price: {1}",
          product.ProductName, product.ProductPrice);
 Console.ReadLine();
```

- The code:
  - Creates a proxy instance of a WCF client of type `ServiceReference1.ProductServiceClient`.
  - The proxy instance is then used to invoke the `GetProductName()` and `GetProductInfo()` service methods.
  - The results returned by the service methods are printed to the console.
- The following figure shows the output of the client application:



Multithreading and Asynchronous Programming

- C# applications often need to execute multiple tasks concurrently.
- For example, a C# application that simulates a car racing game needs to gather user inputs to navigate and move a car while concurrently moving the other cars of the game towards the finish line.
- Such applications, known as multi-threaded applications use multiple threads to execute multiple parts of code concurrently.
- 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 the various classes and interfaces in the `System.Threading` namespace that provides built-in support for multithreaded programming in the .NET Framework.



The Thread Class 1-2

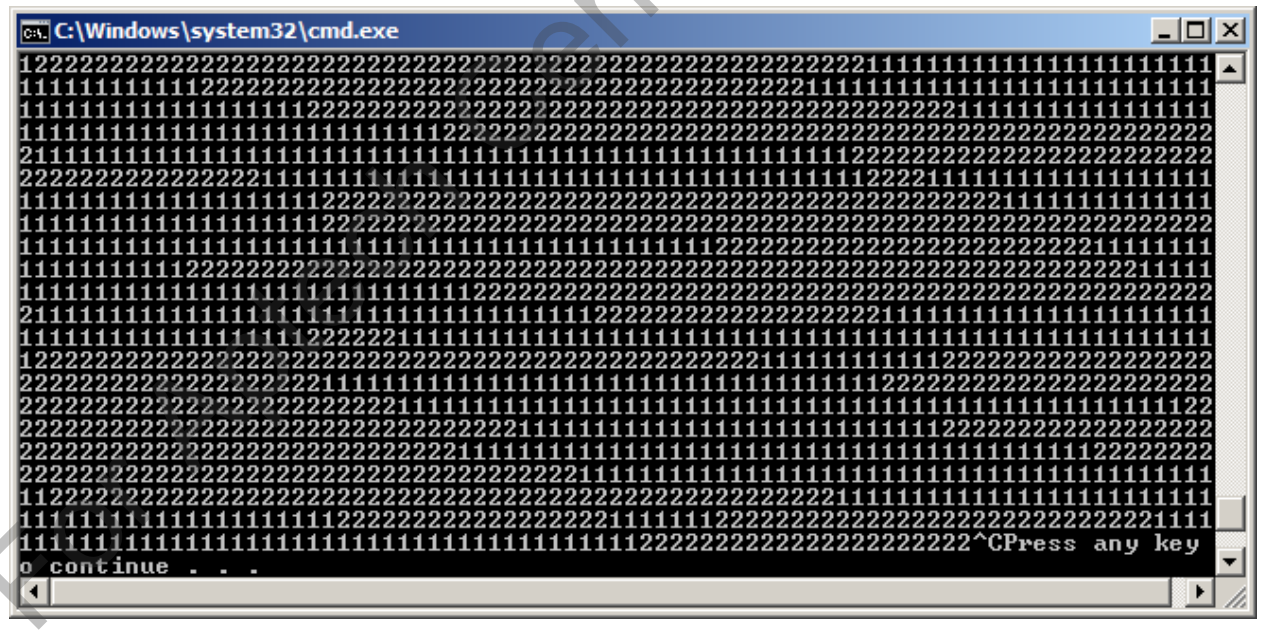
- The `Thread` class of the `System.Threading` namespace allows programmers to create and control a thread in a multithreaded application.
- Each thread in an application passes through different states that are represented by the members of the `ThreadState` enumeration.
- A new thread can be instantiated by passing a `ThreadStart` delegate to the constructor of the 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.
- The following code instantiates and starts a new thread:

Snippet

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

- In the code:
  - The **`Print`**`()` method uses an infinite while loop to print the value `1` to the console. The `Main()` method instantiates and starts a new thread to execute the **`Print`**`()` method.
  - The `Main()` method then uses an infinite while loop to print the value `2` to the console.
  - In this program two threads simultaneously executes both the infinite `while` loops, which results in the output as shown in the following figure:

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 needs to execute in a separate thread of the thread pool.

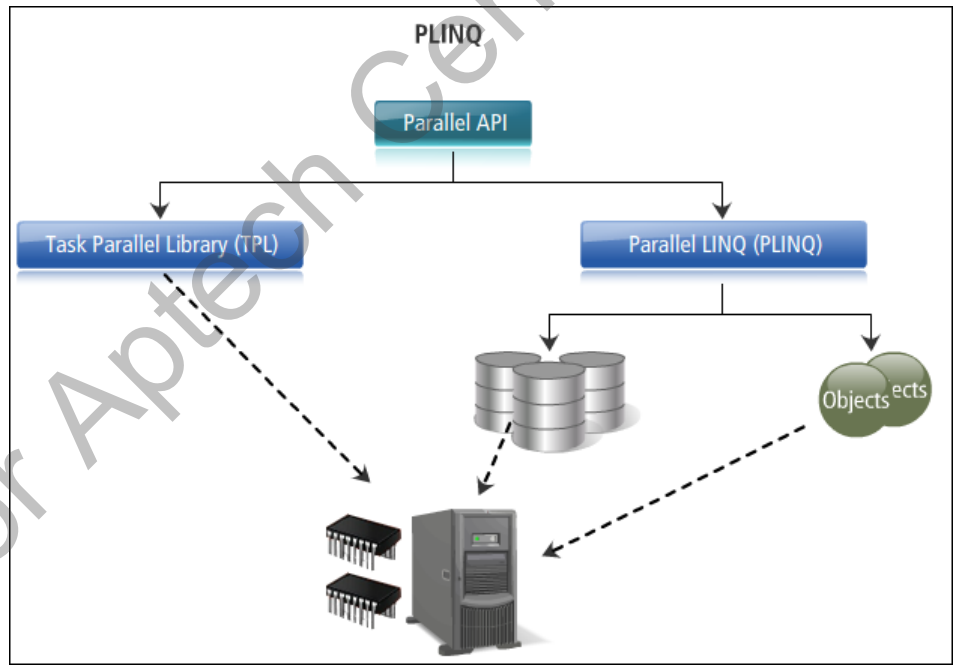
Synchronizing Threads 1-2

- When multiple threads need to share data, their activities needs to 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 needs to 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.

- **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.

- Modern computers contain multiple CPUs. In order to take advantage of the processing power that computers with multiple CPUs deliver, a C# application needs to 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 1-2

- TPL provides the `Task` class in the `System.Threading.Tasks` namespace represents an asynchronous task in a program. Programmers can use this class to invoke a method asynchronously.
- To create a task, the programmer provides a user delegate that encapsulates the code that the task will execute. The delegate can be a named delegate, such as the `Action` delegate, an anonymous method, or a lambda expression. After creating a `Task`, the programmer calls the `Start()` method to start the task.
- 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.
- To ensure that all the tasks of a program completes, 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.
- The following code creates and starts two tasks:

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");
 }
}
```

- In the code:
  - The `Main()` method creates a Task, named `task1` using an `Action` delegate and passing the name of the method to execute asynchronously.
  - The `Start()` method is called to start the task. The `Run()` method is used to create and start another task, named `task2`. The call to the `Wait()` method ensures that both the tasks complete before the `Main()` method exits.

Output

```
Executed by a Task
Executed by a Task
Exiting main method
```

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.

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 needs to 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 1-3

- 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.

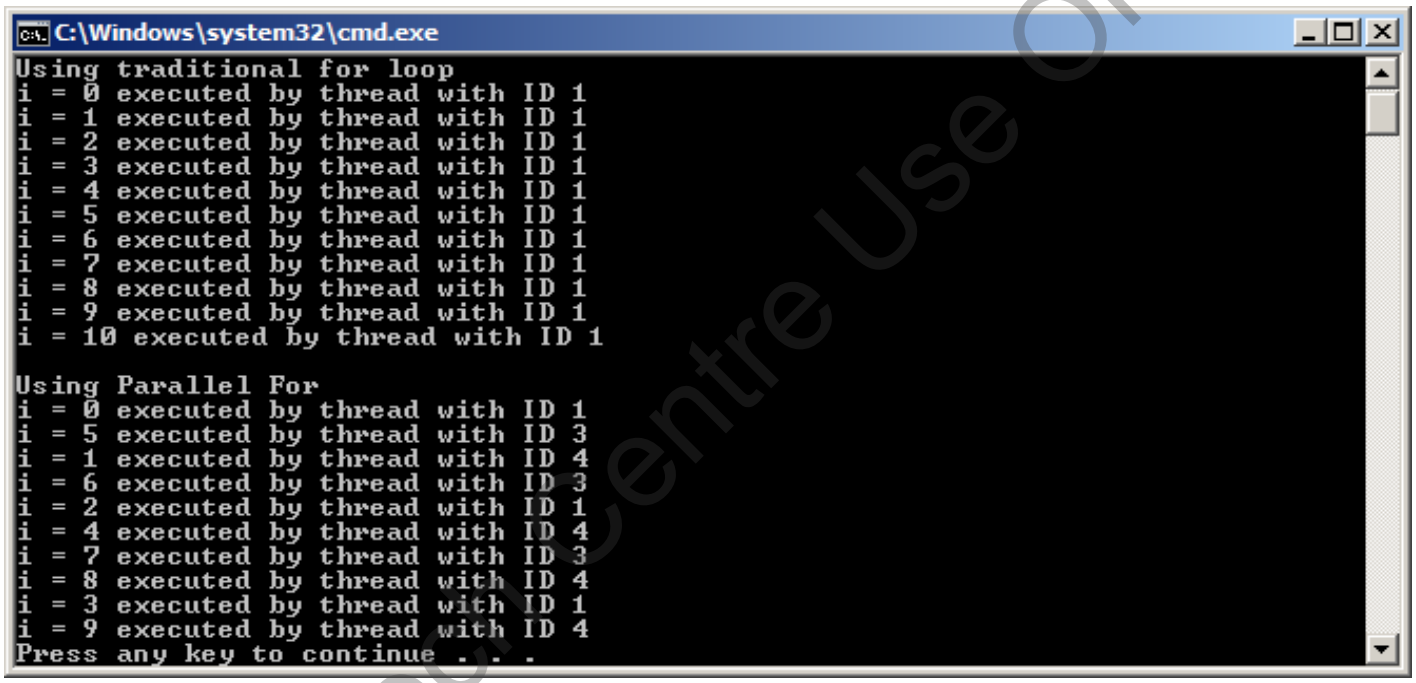
- The following code uses a traditional for loop and the `Parallel.For()` method:

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);
   });
 }
```

- In the code:
  - The `Main()` method first uses a traditional `for` loop to print the identifier of the current thread to the console.
  - The `Sleep()` method is used to pause the main thread for `100` ms for each iteration.
  - As shown in the figure, the `Console.WriteLine()` method prints the results sequentially as a single thread is executing the `for` loop.
  - The `Main()` method then performs the same operation using the `Parallel.For()` method.
  - As shown in the figure multiple threads indicated by the `Thread.CurrentThread.ManagedThreadId` property executes the `for` loop in parallel and the sequence of iteration is unordered.

- The following figure shows one of the possible outputs of the code:



Parallel LINQ (PLINQ) 1-2

- 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.
- For parallel access, PLINQ partitions the data source into segments, and then executes each segment through separate threads in parallel.
- The `ParallelEnumerable` class of the `System.Linq` namespace provides methods that implement PLINQ functionality.
- The following code shows using both a sequential LINQ to Object and PLINQ to query an array:

Snippet

```
string[] arr = new string[] { "Peter","Sam",
"Philip","Andy","Philip","Mary","John","Pamela"};
  var query = from string name in arrselect 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);
    }
```

- The code:
  - Creates a string array initialized with values.
  - A sequential LINQ query is used to retrieve the values of the array that are printed to the console in a `for each` loop.
  - The second query is a PLINQ query that uses the `AsParallel`() method in the form clause.
  - The PLINQ query also performs the same operations as the sequential LINQ query.
  - However, as the PLINQ query is executed in parallel the order of elements retrieved from the source array is different.

Output

```
Names retrieved using sequential LINQ
 Peter
 Sam
 Philip
 Andy
 Philip
 Mary
 John
 Pamela
 Names retrieved using PLINQ
 Peter
 Philip
 Sam
 Mary
 Philip
 John
 Andy
 Pamela
```

Concurrent Collections 1-3

- 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 needs to 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.
- The important classes of the `System.Collections.Concurrent` namespace are as follows:

`ConcurrentDictionary<TKey, 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.

- The 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());
    }
  }
```

- The code:
  - Calls the `TryAdd()` method of the `ConcurrentDictionary` class to concurrently add elements to a `ConcurrentDictionary<string, int>` object using two separate threads.
  - The `TryAdd()` method, unlike the `Add()` method of the `Dictionary` class does not throw an exception if a key already exists.
  - The `TryAdd()` method instead returns false if a key exist and allows the program to exit normally, as shown in the following figure:

Asynchronous Methods 1-3

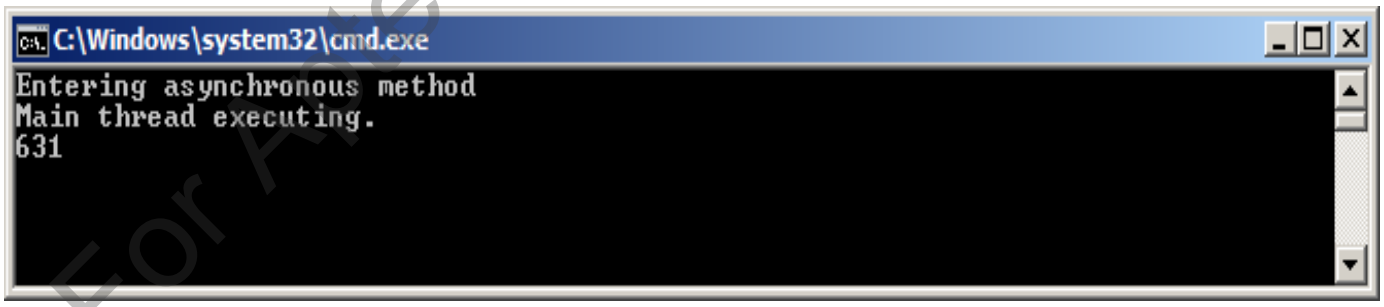
- TPL provides support for asynchronous programming through two new keywords: `async` and `await`.
- These keywords can be used to asynchronously invoke long running methods in a program.
- A method marked with the `async` keyword notifies the compiler that the method will contain at least one await keyword.
- If the compiler finds a method marked as `async` but without an `await` keyword, it reports a compilation error.
- 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 either one of the following return types:
  - `void`
  - `Task`
  - `Task<TResult>`

- The following code shows the use of the `async` and `await` keywords:

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);
   }
 }
```

- In the code:
  - The `AnalyzeData()` method of the `ComplexTask` class creates and starts a new task to execute the `GetResult()` method.
  - The `GetResult()` method simulates a long running operation by making the thread sleep for two seconds before returning a random number.
  - In the `AsyncAwaitDemo` class, the `PerformComputationAsync()` method is marked with the `async` keyword.
  - This method uses the `await` keyword to wait for the `AnalyzeData()` method to return.
  - While waiting for the `AnalyzeData()` method to return, execution is returned to the calling `Main()` method that prints the message "`Main thread executing`" to the console.
  - Once the `AnalyzeData()` method returns, execution resumes in the `PerformComputationAsync()` method and the retrieved random number is printed on the console.
- The following figure shows the output:

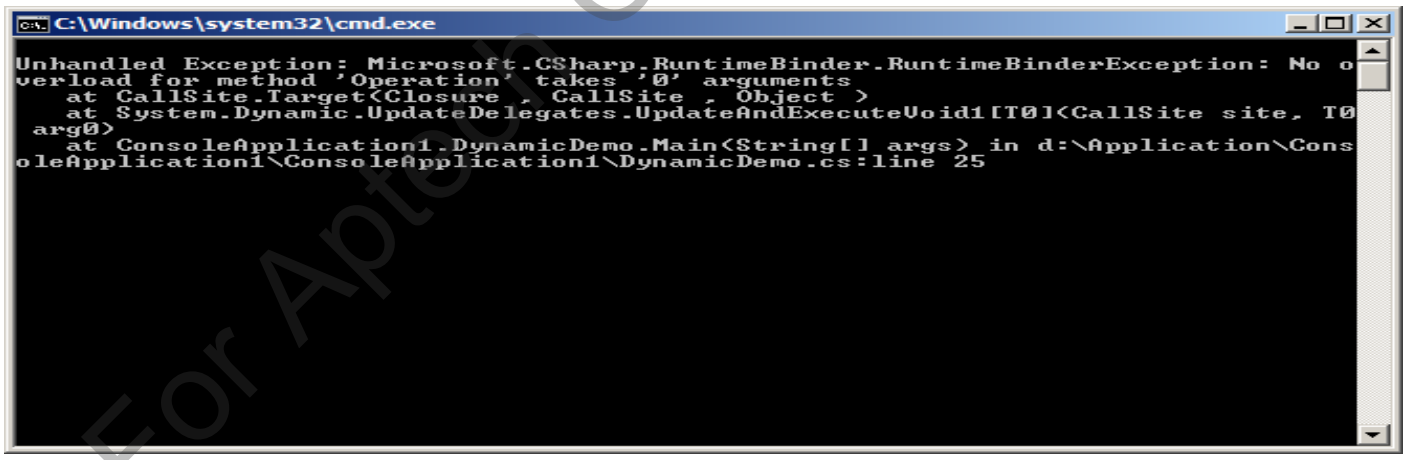


- C# provides dynamic types to support dynamic programming for interoperability of .NET applications with dynamic languages, such as IronPython and COM APIs such as the Office Automation APIs.
- The C# compiler does not perform static type checking on objects of a dynamic type. The type of a dynamic object is resolved at runtime using the Dynamic Language Runtime (DLR).
- 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();

   }
   }
```

- In the code:
  - The **DemoClass** class has a single **Operation()** method that accepts a `String` parameter.
  - The `Main()` method in the **DynamicDemo** class creates a dynamic type and assigns a **DemoClass** object to it.
  - The dynamic type then calls the **Operation()** method without passing any parameter.
  - However, the program compiles without any error as the compiler on encountering the `dynamic` keyword does not performs any type checking.
  - However, on executing the program, a runtime exception will be thrown, as shown in the following figure:



- The `dynamic` keyword can also be applied to fields, properties, method parameters, and return types.
- The following code shows how the `dynamic` keyword can be applied to methods to make them reusable in a program:

Snippet

```
class DynamicDemo {
  static dynamic DynaMethod(dynamic param) {
   if (param is int){
   Console.WriteLine("Dynamic parameter of type int has value
{0}",param);
       return param;
     }
     else if (param is string) {
      Console.WriteLine("Dynamic parameter of type string has
value {0}", param);
       return param;
     }
     else {
      Console.WriteLine("Dynamic parameter of unknown type has
value {0}", param);
       return param;
      }

    }

   static void Main(string[] args) {
    dynamic dynaVar1=DynaMethod(3);
    dynamic dynaVar2=DynaMethod("Hello World");
    dynamic dynaVar3 = DynaMethod(12.5);
    Console.WriteLine("\nReturned dynamic values:\n{0} \n{1} \n{2}", dynaVar1,
dynaVar2, dynaVar3);

    }
  }
```

- In the code, the **`DynaMethod`**`()` method accepts a dynamic type as a parameter. Inside the **`DynaMethod`**`()` method, the `is` keyword is used in an `if-else-if-else` construct to check for the parameter type and accordingly returns a value.
- As the return type of the `DynaMethod()` method is also dynamic, there is no constraint on the type that the method can return.
- The `Main()` method calls the **`DynaMethod`**`()` method with integer, string, and decimal values and prints the return values on the console.

Output

```
Dynamic parameter of type int has value 3
Dynamic parameter of type string has value Hello World
Dynamic parameter of unknown type has value 12.5
Returned dynamic values:
3
Hello World
12.5
```

Summary

- System-defined generic delegates take a number of parameters of specific types and return values of another type.
- A lambda expression is an inline expression or statement block having a compact syntax and can be used in place of a delegate or anonymous method.
- A query expression is a query that is written in query syntax using clauses such as from, select, and so forth.
- 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.
- WCF is a framework for creating loosely-coupled distributed application based on Service Oriented Architecture (SOA).
- Various classes and interfaces in the `System.Threading` namespace provide built-in support for multithreaded programming in the .NET Framework.
- To make parallel and concurrent programming simpler, the .NET Framework introduced TPL, which is a set of public types and APIs in the `System.Threading` and `System.Threading.Tasks` namespaces.