Util Multithreading

Objective

- Introduction to Delegates
- Delegates
- Using lambda in Delegates
- Attribute
- Types of Attributes
- Predefined Attributes
- Obsolete Attribute
- Creating Custom Attributes
- Declaring a Custom Attribute
- Multithreading
- Thread Life Cycle
- Thread class
- Foreground and Background Threads?
- Synchronization
- Thread Synchronization Techniques
- Lock Keyword
- Task Parallel Library (TPL)
- Task
- Serialization
- XML serialization

CASE STUDY - CONTINUED

Case Study (continued)

- As a Shopon admin, I want to give discount on all the products which belongs to Apple company. I should also have flexibility to give discount based on Category too.

Thought

- One of the way to achieve this is using Delegate

Knowledge Byte

Solution

Introduction to Delegates

- Delegates are **type safe function pointer.**
- The signature of the delegate must match the signature of the function, the delegate points to, otherwise we get a compile time error. This is the reason delegates are called as type safe function pointers.
- Delegates are used to pass methods as arguments to other methods.
- Event handlers are nothing more than methods that are invoked through delegates.
- Delegates allow methods to be passed as parameters.
- Delegates can be used to define callback methods.

Delegates

- An objectified function
  - inherits from System.Delegate
  - sealed implicitly
  - looks much like C/C++ style function pointer
- eg. `delegate int Func(ref int x)`
  - defines a new type: Func: takes int, returns int
  - declared like a function with an extra keyword
  - stores a list of methods to call

Simple Example | Complex Example

Delegates - Example

```
public delegate void HelloDelegate(string message);
0 references
class DelegateDemo
{
    1 reference
    public static void Display(string strMessage)
    {
        Console.WriteLine(strMessage);
    }

    0 references
    public static void Main()
    {
        var helloDelegate = new HelloDelegate(Display);
        helloDelegate("This is so nice");
    }
}
```

Complex - Example

```
class MyEmployee
{
    5 references
    public string Name { get; set; }
    5 references
    public double Salary { get; set; }
    4 references
    public int NoExperience { get; set; }

    1 reference
    public void PromoteEmployee(List<MyEmployee> employees, IsPromotable canPromote)
    {
        foreach (var employee in employees)
        {
            if (canPromote(employee))
            {
                Console.WriteLine($"{employee.Name} is promoted");
            }
        }
    }
    delegate bool IsPromotable(MyEmployee myEmployee);
}
```

Complex - Example

```
class EmployeeDelegateDemo
{
    0 references
    static void Main()
    {
        List<MyEmployee> employees = new List<MyEmployee>()...;
        IsPromotable promotable = new IsPromotable(Promot);
        MyEmployee employee = new MyEmployee();
        employee.PromoteEmployee(employees, promotable);

    }

    1 reference
    public static bool Promot(MyEmployee emp)
    {
        if(emp.Salary > 6000)
        {
            return true;
        }
        return false;
    }
}
```

Using Lambda

Using lambda in Delegates

- Instead of creating the function to pass in the delegate, we can create lambda express

```
MyEmployee employee = new MyEmployee();
employee.PromoteEmployee(employees, emp => emp.NoExperience > 7);
```

Solution

```
public class ProductDelegateDemo
{
    private List<Product> products = new List<Product>();
    0 references
    public void Main()
    {
        InitProducts();
        IsDiscountable isDiscountable = new IsDiscountable(Discount);
        var product = new Product();
        var discountedProducts = product.Discount(products, isDiscountable);
        Console.WriteLine("Discounted products");
        foreach (var p in discountedProducts)
        {
            Console.WriteLine($"{p.ProductName}\t{p.Price}");
        }

        Console.WriteLine("-------------------------------------");
        var discountByCategory = product.Discount(products, x => x.CategoryName.Equals("IOS"));
        Console.WriteLine("IOS products");
        Console.WriteLine("-------------------------------------");
        foreach (var p in discountByCategory)
        {
            Console.WriteLine($"{p.ProductName}\t{p.Price}");
        }
    }
```

```
    private  bool Discount(Product product)
    {
        if (product.Price > 30000)
        {
            return true;
        }
        return false;
    }

    1 reference
    private  void InitProducts()
    {
        products.Add(new Product()...);
        products.Add(new Product()...);
        products.Add(new Product()...);
        products.Add(new Product()...);
        products.Add(new Product()...);
        products.Add(new Product()...);
        products.Add(new Product()...);
        products.Add(new Product()...);
        products.Add(new Product()...);
        products.Add(new Product()...);
    }
}
```

Solution

```
public delegate bool IsDiscountable(Product product);
19 references
public class Product
{
    10 references
    public int Id { get; set; }
    12 references
    public string ProductName { get; set; }
    11 references
    public string CategoryName { get; set; }
    10 references
    public string CompanyName { get; set; }
    13 references
    public double Price { get; set; }
    10 references
    public string ImageUrl { get; set; }

    2 references
    public IEnumerable<Product> Discount(List<Product> products, IsDiscountable discountable)
    {
        List<Product> newProducts = new List<Product>();
        foreach (var product in products)
        {
            if (discountable(product))
            {
                newProducts.Add(product);
            }
        }
        return newProducts;
    }
}
```

Case Study (continued)

- As a Shopon architect, I want to deprecate **Get(int id)** method in ProductRepository as we have overload method **Get(string key)** where we can extend the functionality.

Thought

- As we must deprecate or **Obsolete** the method, we can use **Attribute Obsolete** for the method.

Knowledge Byte

Solution

Attribute

- An **attribute** is a declarative tag that is used to convey information to runtime about the behaviors of various elements like classes, methods, structures, enumerators, assemblies etc. in our program.
- We can add declarative information to a program by using an attribute.
- A declarative tag is depicted by square **([ ])** brackets placed above the element it is used for.
- Attributes are used for adding metadata, such as compiler instruction and other information such as comments, description, methods and classes to a program.

Types of Attributes

- The .Net Framework provides two types of attributes:
  - *the pre-defined* attributes and
  - *custom built* attributes

- Syntax:

```
[attribute(positional_parameters, name_parameter = value, ...)]
element
```

Name of the attribute and its **values** are specified within the square brackets, before the element to which the attribute is applied. Positional parameters specify the essential information and the name parameters specify the optional information.

Predefined Attributes

- The .NET Framework provides three pre-defined attributes –

  – AttributeUsage - The pre-defined attribute **AttributeUsage** describes how a custom attribute class can be used. It specifies the types of items to which the attribute can be applied.

  – Conditional - This predefined attribute marks a conditional method whose execution depends on a specified preprocessing identifier.

  – Obsolete - This predefined attribute marks a program entity that should not be used. It enables us to inform the compiler to discard a particular target element.

Version 1.0

Obsolete Attribute

Syntax

```
[Obsolete (
   message
)]

[Obsolete (
   message,
   iserror
)]
```

Example

```
class AttributeDemo
{
    [Obsolete("Don't use OldMethod, use NewMethod instead", false)]

    1 reference
    static void OldMethod()
    {
        Console.WriteLine("It is the old method");
    }
    0 references
    static void NewMethod()
    {
        Console.WriteLine("It is the new method");
    }
    0 references
    public static void Main()
    {
        OldMethod();
    }
}
```

Creating Custom Attributes

- The .NET Framework allows creation of custom attributes that can be used to store declarative information and can be retrieved at run-time.

- This information can be related to any target element depending upon the design criteria and application need.

- Creating and using custom attributes involve four steps –
  - Declaring a custom attribute
  - Constructing the custom attribute
  - Apply the custom attribute on a target program element
  - Accessing Attributes Through Reflection

Declaring a Custom Attribute

- A new custom attribute should be derived from the **System.Attribute** class.

```
//a custom attribute BugFix to be assigned to a class and its members
[AttributeUsage(
   AttributeTargets.Class |
   AttributeTargets.Constructor |
   AttributeTargets.Field |
   AttributeTargets.Method |
   AttributeTargets.Property,
   AllowMultiple = true)]

public class DeBugInfo : System.Attribute
```

Version 1.0

Example

```
//a custom attribute BugFix to be assigned to a class and its members
[AttributeUsage(
   AttributeTargets.Class |
   AttributeTargets.Constructor |
   AttributeTargets.Field |
   AttributeTargets.Method |
   AttributeTargets.Property,
   AllowMultiple = true)]

public class DeBugInfo : System.Attribute {
   private int bugNo;
   private string developer;
   private string lastReview;
   public string message;

   public DeBugInfo(int bg, string dev, string d) {
      this.bugNo = bg;
      this.developer = dev;
      this.lastReview = d;
   }
   public int BugNo {
      get {
         return bugNo;
      }
   }
   public string Developer {
      get {
         return developer;
      }
   }
   public string LastReview {
      get {
         return lastReview;
      }
   }
   public string Message {
      get {
         return message;
      }
      set {
         message = value;
      }
   }
}
```

Example

```
[DeBugInfo(45, "Saheer", "12/8/2021", Message = "Return type mismatch")]
[DeBugInfo(49, "Shashi", "10/10/2021", Message = "Unused variable")]
2 references
class CustomAttributDemo
{
    protected double length;
    protected double width;
    1 reference
    public CustomAttributDemo(double l, double w)
    {
        length = l;
        width = w;
    }

    [DeBugInfo(55, "Shashi", "19/10/2012", Message = "Return type mismatch")]
    1 reference
    public double GetArea()
    {
        return length * width;
    }

    [DeBugInfo(56, "Saheer", "19/10/2012")]
    1 reference
    public void Display()
    {
        Console.WriteLine("Length: {0}", length);
        Console.WriteLine("Width: {0}", width);
        Console.WriteLine("Area: {0}", GetArea());
    }

    0 references
    static void Main()
    {
        new CustomAttributDemo(2, 4).Display();
    }
}
```

Solution

```
public interface IProductRepo
{
    /// <summary> Method to get all products
    4 references | 0/2 passing
    IEnumerable<Product> GetProducts();

    /// <summary> Method to get product based on id
    [Obsolete(message: "This method is obsolete. Use Get(key)")]
    5 references | 0/3 passing
    Product Get(int id);

    /// <summary> Method to get products based on company name or product name
    2 references
    IEnumerable<Product> Get(string key);
}

public class ProductManager : IProductManager
{
    private readonly IProductRepo productRepo;

    1 reference
    public ProductManager(IProductRepo productRepo)...

    #region Public Method
    public Product GetProductById(int id) => this.productRepo.Get(id);

    1 reference
    public IEnumerable<Product> GetProductByKey(string key) => this.productRepo.Get(key);

    2 references | 0/1 passing
    public IEnumerable<Product> GetProducts() => this.productRepo.GetProducts();
    #endregion
}
```

NOTE: In ProductManager, we will get warning as we are using method which is marked Obsolete.

Multitasking

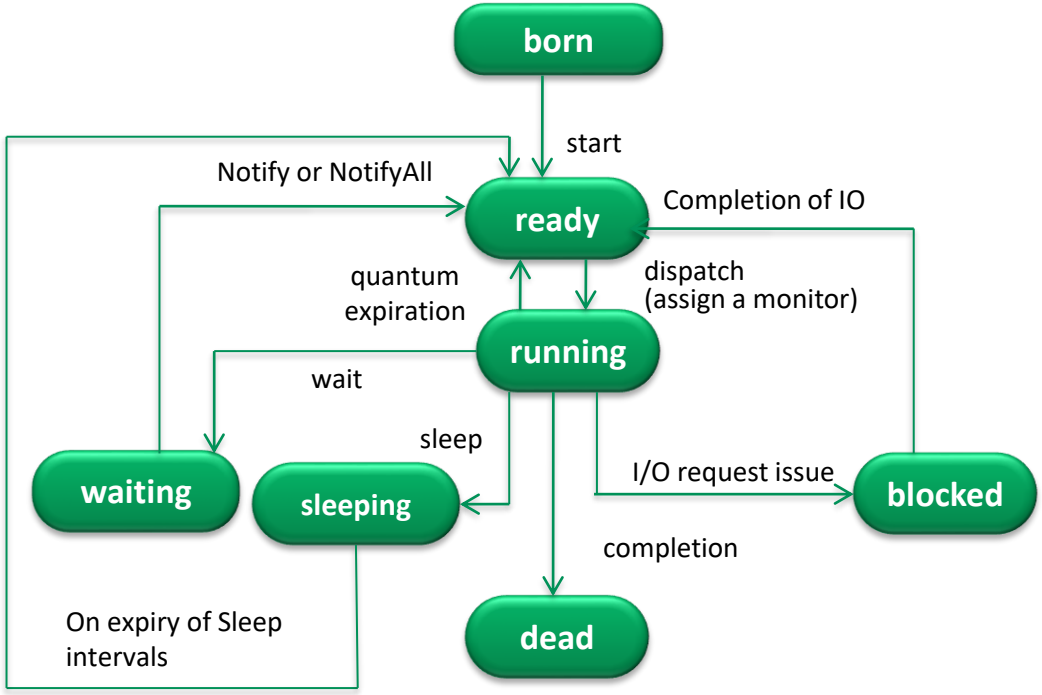
- Every application executes in its own address space or process.
  - E.g. MS Word (Creating a document) or WinAmp (listening songs)
- A processor can execute applications in multiple processes concurrently.
  - Ability to execute different applications or tasks by the processor is called multitasking
    - The scheduling algorithm of the operating system decides it
      - Based on time slicing – pre-emptive
- Creating a document in MS Word and listening to a song in WinAmp can be done concurrently
- Operating System switches between the applications running in different processes
  - Each process has its own data which needs to be stored temporarily before switching to other process. This is called context.
  - Context switch is resource heavy

Multithreading

- CPU is idle between the context switch
- Consider a scenario of a single application - MS Word
  - Spell check, grammar check runs while the editing is being done.
- Ability of OS to run different parts of the same program or application concurrently is called multithreading.
  - Each part of the program is called “Thread”.
  - Thread can execute separate tasks independently with each task not dependent on other.
  - Two or more threads in an application can share same data.
    - Synchronization of threads is required in such a scenario as each thread can change the shared data.

Thread Life Cycle

- Thread is a part of execution of an application. It has a life-cycle.



Thread class

- *System.Threading* namespace provides a set of classes and other types that support in creating multithreaded applications.
- Thread Class
  - Represents a thread, helps to create it, control it, set its priority and get its status .

Method Name	Description
static void Sleep(int millisecondsTimeout)	Suspends the current thread for a particular time interval.
void Start()	Causes the state of thread to be in running state. It is an overloaded method.
void Join()	Blocks the calling thread until the thread terminates
void Abort()	Raises a *ThreadAbortException* in the thread on which it is invoked, to begin the process of terminating the thread. Calling this method usually terminates the thread.
CurrentThread	property; gets the currently running thread
Name	gets or sets the name of the thread
ThreadPriority	Gets or sets a value indicating the scheduling priority of a thread.

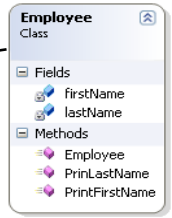
Thread class - Example

A Simple Example of Multithreading

```
static void Main(string[] args)
{
    Employee employee1 = new Employee("Remo", "Mathew");
    ThreadStart delegate1 = new ThreadStart(employee1.PrintFirstName);
    Thread firstThread = new Thread(delegate1);

    ThreadStart delegate2 = new ThreadStart(employee1.PrintLastName);
    Thread secondThread = new Thread(delegate2);

    firstThread.Start();
    secondThread.Start();


    Employee employee2 = new Employee("John", "Hill");
    employee2.PrintFirstName();
    employee2.PrintLastName();

}
```

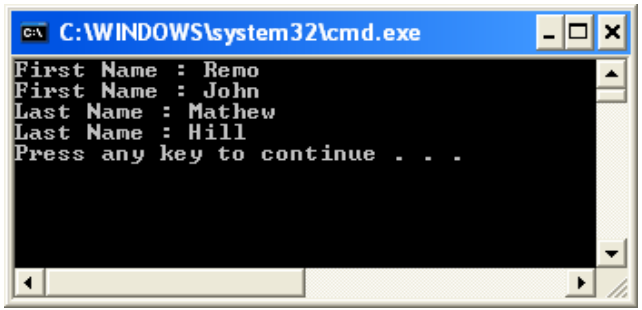
Call to the methods on threads

Normal method call from Main



used by the client code

Output – unpredicted depending on the scheduling algorithm of OS



What are Foreground and Background Threads?

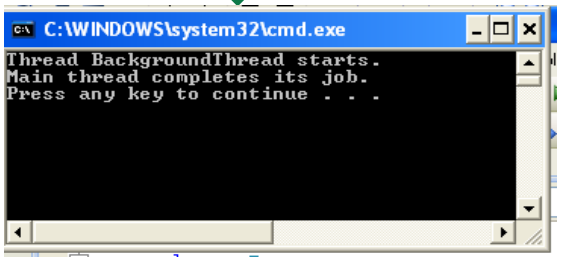
- Thread can be either foreground or background thread.
  - Foreground thread keep the execution environment running.
  - Background thread run in the background. It stops once the foreground thread stops.
- *IsBackround* property of a thread determines whether the thread is background.
  - By default the thread is foreground.
- An application runs on the main thread.
  - If some tasks are time consuming and their execution and output does not affect other threads in the application, such threads are made as background thread.

Foreground and Background Threads - Example

```
static void TaskOnThread()
{
    Console.WriteLine("Thread {0} starts.", Thread.CurrentThread.Name);
    Thread.Sleep(3000);
    Console.WriteLine("Thread {0} ends.", Thread.CurrentThread.Name);

}
```

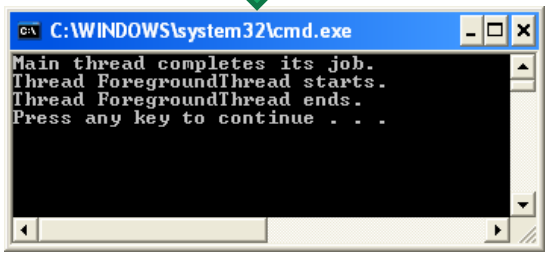
```
Thread t1 = new Thread(TaskOnThread);
t1.Name = "BackgroundThread";
t1.IsBackground = true; //make it as background thread
t1.Start();
Console.WriteLine("Main thread completes its job.");
```

I




```
Thread t1 = new Thread(TaskOnThread);
t1.Name = "ForegroundThread";
t1.IsBackground = false; //foreground thread
t1.Start();
Console.WriteLine("Main thread completes its job.");
```

II



Why Synchronization?

- Two or more threads sharing the same data.
  - Data becomes inconsistent.
- Two important problems
  - Race Condition
    - Two threads try to reach a particular block of code first.
  - Deadlock
    - One thread tries to lock a resource which the other thread has already locked.
- Synchronization of threads should be done to avoid the problems.

Thread Synchronization Techniques

- Synchronization of threads is required when two or more threads share same data.
- Thread Synchronization techniques
  - Synchronization context
    - *[Synchronization]* attribute is used
  - Synchronized Code Regions
    - Using Monitor class
  - Using *lock* keyword in C#
    - Statements which need synchronization are put in a block
  - Manual Synchronization
    - Using *Interlocked* class
- Inter – process Synchronization
  - Using *Mutex* class
- MethodImplAttribute
  - Synchronizes an entire method in one single command

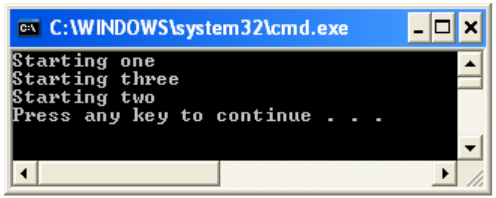
Synchronizing Code Regions - *Monitor* class

- This class helps to control lock on an object for a particular region of the code for a single thread.
  - Region is called critical section.
  - No other thread can access the critical section till the lock is released.
  - Only reference types can be locked and not the value types.
  - **Value type have to be boxed if used**
- Monitor class maintains information
  - Reference to the thread that holds the lock
  - Reference to the threads waiting in the queue to obtain the lock
  - Reference to the queue itself
- Methods of Monitor class
  - *Enter() , TryEnter()*
  - *Wait()*
  - *Pulse(), PulseAll()*
  - *Exit()*

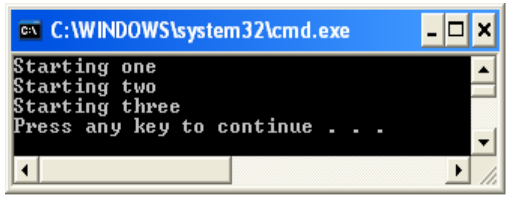
```
class CustomThread
{
    MyStringClass _shared;
    string myString;

    public CustomThread(MyStringClass shared, string str)
    {
        this._shared = shared;
        this.myString = str;
    }
    public void Run()
    {
        Monitor.Enter(_shared);
        try
        {
            _shared.PrintString(myString);
        }
        catch (Exception ex)
        {
            Console.WriteLine(ex.Message);
        }
        Monitor.Exit(_shared);
    }
}
```

Synchronizing using Monitor class

O/P if Monitor class is not used (inconsistent O/P)



O/P when Monitor class is used (synchronization of code region)




```
MyStringClass sharedObject = new MyStringClass();

CustomThread first = new CustomThread(sharedObject,"one");
CustomThread second = new CustomThread(sharedObject, "two");
CustomThread third = new CustomThread(sharedObject, "three");

Thread thread1 = new Thread(new ThreadStart(first.Run));
Thread thread2 = new Thread(new ThreadStart(second.Run));
Thread thread3 = new Thread(new ThreadStart(third.Run));
```

Lock Keyword

- *Lock k*eyword helps to control lock on a particular object for a particular section of code.
  - Uses *Enter()* and *Exit()* methods
  - Simplified version of *Monitor* class

```
class SampleClass
{
    private object object1 = new object();
    private int i;
    public void Method1()
    {
      lock (object1)
      {
         // code which needs locking
         // . . .
         // lock is released automatically
          //after the last executable statement
      }
    }
}
```

Task Parallel Library (TPL)

- TPL is a set of public types and APIs in the **System.Threading** and **System.Threading.Tasks** namespaces.
- TPL has been created in order to help developers to easily add parallelism and concurrency in their .NET applications.

Task

- Represents an asynchronous operation.

```
public class Task : IAsyncResult, IDisposable
```

- The **Task** class represents a single operation that does not return a value and that usually executes asynchronously.
- Task objects are one of the central components of the task-based asynchronous pattern first introduced in the .NET Framework 4. Because the work performed by a Task object typically executes asynchronously on a thread pool thread rather than synchronously on the main application thread, we can use the **Status** property, as well as the **IsCanceled**, **IsCompleted**, and **IsFaulted** properties, to determine the state of a task.
- Most commonly, a lambda expression is used to specify the work that the task is to perform.

Creating Task

- The most commonly used way and which is available starting with the .NET Framework 4.5, is to call the static **Run** method.
- The **Run** method provides a simple way to start a task using default values and without requiring additional parameters.

```
public static async Task Main()
{
   await Task.Run( () => {
                              // Just loop.
                              int ctr = 0;
                              for (ctr = 0; ctr <= 1000000; ctr++)
                              {}
                              Console.WriteLine("Finished {0} loop iterations",
                                                ctr);
                           } );
}
```

- An alternative, and the most common method to start a task in .NET Framework 4, is the static **TaskFactory.StartNew** method.
- The **Task.Factory** property returns a **TaskFactory** object. Overloads of the **TaskFactory.StartNew** method let you specify parameters to pass to the task creation options and a task scheduler.

```
public static void Main()
{
   Task t = Task.Factory.StartNew( () => {
                              // Just loop.
                              int ctr = 0;
                              for (ctr = 0; ctr <= 1000000; ctr++)
                              {}
                              Console.WriteLine("Finished {0} loop iterations",
                                                ctr);
                           } );
   t.Wait();
}
```

Task Creation and Execution

- The **Task.Run** or **TaskFactory.StartNew** method is the preferred mechanism for creating and scheduling computational tasks, but for scenarios where creation and scheduling must be separated, we can use the constructors and then call the **Task.Start** method to schedule the task for execution at a later time.

Task - Wait

- Tasks typically run asynchronously on a thread pool thread, the thread that creates and starts the task continues execution as soon as the task has been instantiated.
- In some cases, when the calling thread is the main application thread, the app may terminate before the task actually begins execution.
- In others, our application's logic may require that the calling thread continue execution only when one or more tasks have completed execution.
- We can synchronize the execution of the calling thread and the asynchronous tasks it launches by calling a **Wait** method to wait for one or more tasks to complete.

Task – Wait (continued)

- To wait for a single task to complete, you can call its **Task.Wait** method.
- A call to the **Wait** method blocks the calling thread until the single class instance has completed execution.
- We can also conditionally wait for a task to complete.
  The **Wait(Int32)** and **Wait(TimeSpan)** methods block the calling thread until the task finishes or a timeout interval elapses, whichever comes first.

Task - Wait

```
using System;
using System.Threading;
using System.Threading.Tasks;

class Program
{
    static Random rand = new Random();

    static void Main()
    {
        // Wait on a single task with no timeout specified.
        Task taskA = Task.Run( () => Thread.Sleep(2000));
        Console.WriteLine("taskA Status: {0}", taskA.Status);
        try {
          taskA.Wait();
          Console.WriteLine("taskA Status: {0}", taskA.Status);
       }
       catch (AggregateException) {
          Console.WriteLine("Exception in taskA.");
       }
    }
}
// The example displays output like the following:
//     taskA Status: WaitingToRun
//     taskA Status: RanToCompletion
```

Creating async Task

```
public async Task<List<Product>> GetProductsAsync()
{
    List<Product> products = new List<Product>();
    string sqlSt = "SELECT Pid, ProductName, Price, CompanyId, CategoryId, AvailableStatus FROM dbo.Product;";
    return await Task.Run(() =>
    {
        using (SqlConnection con = new SqlConnection(connectionString))
        {
            con.Open();
            SqlCommand command = new SqlCommand(sqlSt, con);
            SqlDataReader reader = command.ExecuteReader();
            while (reader.Read())
            {
                Product product = new Product();
                product.ProductId = Convert.ToInt32(reader["PId"]);
                product.ProductName = reader["ProductName"].ToString();
                product.Price = Convert.ToInt32(reader["Price"]);
                product.CompanyId = Convert.ToInt32(reader["CompanyId"]);
                product.CategoryId = Convert.ToInt32(reader["CategoryId"]);
                product.AvailableStatus = reader["AvailableStatus"].ToString();
                products.Add(product);
            }
        }
        return products;
    });
}
```

Calling async method

```
public async Task<IEnumerable<Product>> GetProductsAsyncValues()
{
    return await GetProductsAsync();
}

var productsAsync = app.GetProductsAsyncValues().Result;
foreach (var product in productsAsync)
{
    Console.WriteLine($"{product.ProductId}\t{product.ProductName}");
}
```

NOTE: .Result will convert the Task to the result expected.

XML Serialization

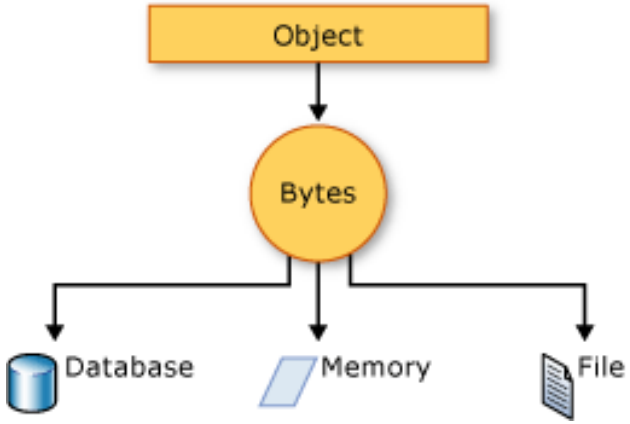
Serialization

- **Serialization** is the process of *converting an object into a stream of bytes* to store the object or transmit it to memory, a database, or a file.

- Its main purpose is to save the state of an object in order to be able to recreate it when needed. The reverse process is called **deserialization**.

How serialization works

- The object is serialized to a stream that carries the data. The stream may also have information about the object's type, such as its version, culture, and assembly name. From that stream, the object can be stored in a database, a file, or memory.



Uses for serialization

- Serialization allows the developer to save the state of an object and re-create it as needed, providing storage of objects as well as data exchange.

- Through serialization, a developer can perform actions such as:
  - Sending the object to a remote application by using a web service
  - Passing an object from one domain to another
  - Passing an object through a firewall as a JSON or XML string
  - Maintaining security or user-specific information across applications

XML serialization

- XML serialization serializes the *public fields* and *properties* of an object, or the parameters and return values of methods, into an **XML stream** that conforms to a specific XML Schema definition language (XSD) document.

- XML serialization results in strongly typed classes with public properties and fields that are converted to XML.

- **System.Xml.Serialization** contains classes for serializing and deserializing XML. You apply attributes to classes and class members to control the way the **XmlSerializer** serializes or deserializes an instance of the class.

Serialize DataSet

```
private void SerializeDataSet(string filename)
{
    XmlSerializer ser = new XmlSerializer(typeof(DataSet));

    // Creates a DataSet; adds a table, column, and ten rows.
    DataSet ds = new DataSet("myDataSet");
    DataTable t = new DataTable("table1");
    DataColumn c = new DataColumn("thing");
    t.Columns.Add(c);
    ds.Tables.Add(t);
    DataRow r;
    for (int i = 0; i < 10; i++)
    {
        r = t.NewRow();
        r[0] = "Thing " + i;
        t.Rows.Add(r);
    }
    TextWriter writer = new StreamWriter(filename);
    ser.Serialize(writer, ds);
    writer.Close();
}
static void Main()
{
    new SerializeDataSetDemo().SerializeDataSet("c:/temp/a.xml");
}
```

Serializing an XmlElement and XmlNode

```
private void SerializeElement(string filename)
{
    XmlSerializer ser = new XmlSerializer(typeof(XmlElement));
    XmlElement myElement =
    new XmlDocument().CreateElement("MyElement", "ns");
    myElement.InnerText = "Hello World";
    TextWriter writer = new StreamWriter(filename);
    ser.Serialize(writer, myElement);
    writer.Close();
}

private void SerializeNode(string filename)
{
    XmlSerializer ser = new XmlSerializer(typeof(XmlNode));
    XmlNode myNode = new XmlDocument().
    CreateNode(XmlNodeType.Element, "MyNode", "ns");
    myNode.InnerText = "Hello Node";
    TextWriter writer = new StreamWriter(filename);
    ser.Serialize(writer, myNode);
    writer.Close();
}
static void Main()
{
    new SerializeNodeElementDemo().SerializeElement(@"c:\temp\SerializeElement.xml");
    new SerializeNodeElementDemo().SerializeNode(@"c:\temp\SerializeNode.xml");
}
```

Next Step

Recap

Useful links

Thank you

Thank YOU

- Any Questions?

US – CORPORATE HEADQUARTERS

1248 Reamwood Avenue,
Sunnyvale, CA 94089
Phone: (408) 743 4400

343 Thornall St 720
Edison, NJ 08837
Phone: (732) 395 6900

UK

57 Rathbone Place,
4th Floor, Holden House,
London, W1T 1JU , UK

89 Worship Street Shoreditch,
London EC2A 2BF, UK
Phone: (44) 2079 938 955

INDIA

Mumbai
4th Floor, Nomura
Powai , Mumbai 400 076
Phone: +91 (22) 3051 1000

Pune
5th floor, Amar Paradigm Baner Road
Baner, Pune 411 045
Phone: +91 (20) 6604 6000

Bangalore
4th Floor, Kabra Excelsior,
80 Feet Main Road, Koramangala 1st Block,
Bengaluru (Bangalore) 560034
Phone: +91 (80) 4666 1666

www.xoriant.com