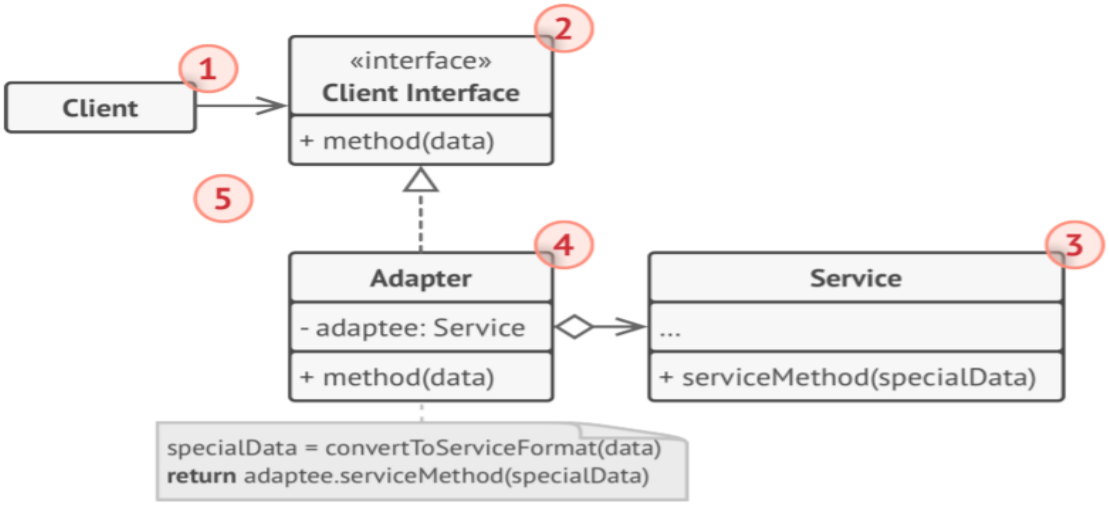
Adapter Pattern Implementation Lab Task 1

Introduction & Concept

In this lab task we will learn how to implement the adapter pattern using C#.Net

- Adapter pattern converts the interface of a class into another interface that clients expect. It lets classes work together that could not otherwise because of incompatible interfaces.
- The following is the structure of prototype pattern.



Implementation

1. Create a console application in visual studio.
2. Create a folder and name it AdapterImplementation
3. Create ***IClient.CS*** file in this folder and write the following code in it:

```
/// <summary>
/// Adapter Client Interface
```

```
/// </summary>
public interface IClient
{
    void Send(string data);
}
```

The above created interface is an abstraction that declares a single method Send(). It will be used by the clients that are incompatible to work otherwise.

In the next step we are going to create a service class that will receives the data from the client but has a incompatible interface, means that the data sent by the client is not in the format that this service requires.

4. Create **Service.cs** file in this folder and write the following code in it:

```
/// <summary>
/// Service class receives the integer data
/// </summary>
public class Service
{
    public void Recieve(int data)
    {
       Console.WriteLine($"Reached to the service {data}");
    }
}
```

The above service class has incompatible interface that the client expects. So we need an adapter to fix this incompatibility interface issue.

Next, we are going to create an adapter.

5. Create **Adapter.cs** file in this folder and write the following code in it:

```
/// <summary>
/// Adpater implementing the IClient
/// </summary>
public class Adapter : IClient
{
    Service service { get; set; } = new Service();
```


```
        public void Send(string data)
        {
            Console.WriteLine("Data received from client...");
            var x = Convert.ToInt32(data);
            service.Recieve(x);
        }
    }
```

The adapter implementation converts the client data into the format that a service class expects and thus resolves the problem. Now the clients can send data in string format that will automatically be converted into the integer format by the adapter and sent to the service. Here we have accomplished the adapter implementation task.

Next, we are going to create a adapter demo class to demonstrate how incompatible interface work together.

6. Create ***AdapterDemo.cs*** file in this folder and write the following code in it:

```
    /// <summary>
    /// Using adapter
    /// </summary>
    public class AdapterDemo
    {
        public static void Main(string[] args)
        {
            IClient client = new Adapter();
            client.Send("900");

            Console.ReadKey();
        }

    }
```

In the above demo class we created an adapter instance and store its reference in IClient interface, then we call the adapters send method to send data in the string format that is incompatible to the service class but the adapter will take care of it automatically.
The purpose of the Adapter Pattern is to allow different classes work together that have incompatible interfaces and in this lab task we demonstrated the problems scenario that can be resolved using adapter pattern.
==========END TASK==========

Prototype Pattern Implementation Lab Task 2

In this lab task we are going to solve a problem similar in context as demonstrated in Lab Task 1 but a bit complex. This example is comprehensive in order to solve the real problem.

The Problem

Let's imagine that we have functionality in which we convert the list of students into JSON format and use it in our application code given below. But instead of a list, we have been provided with an API (Student Data Provider) that provides list of students in XML format.

Assume that we can't modify the existing API functionality (because of the technical restrictions such as the API being imported into our project from another solution or as a NuGet package that is written by a third party.

Considering above restrictions, we must find a way around it and that proper way is to implement the Adapter design pattern to solve this type of problem.

Existing Application Code

Following is the existing application code, given to you for your understanding to implement the Adapter Design Pattern. You are not allowed to modify it but use it.

Create a new console project in Visual Studio and create a new folder on the root, name it Adapter.
Create the following class files in this folder.

1. Student Model – *Student.cs*

```
public class Student
{
    public int Id { get; set; }
    public string Name { get; set; }
    public string Email { get; set; }
}
```


2. Student Data Provider : *StudentDataProvider.cs*

```
//Student Data Provider
 public static class StudentDataProvider
 {
     public static List<Student> GetData() =>
        new List<Student>
        {
             new Student { Id = 100, Name = "Altaf Dar",
                     Email="altaf.dar@gmail.com" },
             new Student { Id = 101, Name = "Martin Minro",
                     Email="martin.minro@yahoo.com" },
             new Student { Id = 102, Name = "Sara Dasner",
                     Email="sara.daner@microsoft.com" },
             new Student { Id = 103, Name = "Arika Gillani",
                     Email="arika.gillani@gmail.com" },
             new Student { Id = 104, Name = "Amjad Kahn",
                     Email="amjad.khan@outlook.com" }
        };
 }
```

3. XML Converter : *XmlConverter.cs*

```
public class XmlConverter
    {
        public XDocument GetXML()
        {
            var xDocument = new XDocument();
            var xElement = new XElement("Students");
            var xAttributes = StudentDataProvider.GetData()
                .Select(s => new XElement("Student",
                                new XAttribute("Id", s.Id),
                                new XAttribute("Name", s.Name),
                                new XAttribute("Email", s.Email)));

            xElement.Add(xAttributes);
            xDocument.Add(xElement);
            return xDocument;
        }
    }
```

4. JSON Converter : *JsonConverter.cs*

```
    public class JsonConverter
    {
        IEnumerable<Student> Students;
        public JsonConverter(IEnumerable<Student> students)
        {
            Students = students;
        }
        public string ToJson()
        {
            var jsonResult = JsonConvert.SerializeObject(
                      Students, Formatting.Indented);
            return jsonResult;
        }
    }
```

5. Demo Without Adapter : *Demo.cs*

```
public static class Demo
{
    public static void Main(string[] args)
    {
        var xml = new XmlConverter();
        Console.WriteLine(xml.GetXML());
        Console.WriteLine();

        var students = StudentDataProvider.GetData();
        var json = new JsonConverter(students);
        Console.WriteLine(json.ToJson());

        Console.ReadKey();
    }
}
```

Now you can have a clear understating of the above existing application code. Next we will implement the adapter design pattern to solve the incompatible interface problem without modifying the existing code. Lets do it...

Adapter Implementation

As we can see, there is no way to pass xml data returned from the `GetXml()` to the `JsonConverter` class, so we need to implement adapter design pattern which will make these two interfaces work together.

The task is to create the adapter functionality so that the above incompatible interfaces will work together.

6. XmlToJson Adapter Interface : *IXmlToJson.cs*

```
//Implementation of Adapter Pattern
 public interface IXmlToJson
 {
     string ConvertXmlToJson();
 }
```


7. XmlToJsonAdapter Interface : *XmlToJsonAdapter.cs*

```
public class XmlToJsonAdapter : IXmlToJson
{
    XmlConverter XmlConverter { get; }
    public XmlToJsonAdapter( XmlConverter xmlConverter)
    {
        XmlConverter = xmlConverter;
    }
    public string ConvertXmlToJson()
    {
        var students = XmlConverter.GetXML()
             .Element("Students")
             .Elements("Student")
             .Select(s => new Student
             {
                Id = Convert.ToInt32(s.Attribute("Id").Value),
                Name = s.Attribute("Name").Value,
                Email = s.Attribute("Email").Value
             });
        var jsonConverter = new JsonConverter(students);
        return jsonConverter.ToJson();
    }
}
```

9. Demo Using XmlToJsonAdapter : *Demo.cs*

```
public static class Demo
 {
     public static void Main(string[] args)
     {
         var xmlConverter = new XmlConverter();
         var adapter = new XmlToJsonAdapter(xmlConverter);
         var json = adapter.ConvertXmlToJson();
         Console.WriteLine(json);
         Console.ReadKey();
     }
 }
```

By implementing adapter Design Pattern in out code above we are able to resolve the incompatible interface problem between two components or classes without modifying the existing application code.

Goodbye, wish you all the best and see you in next lab task!