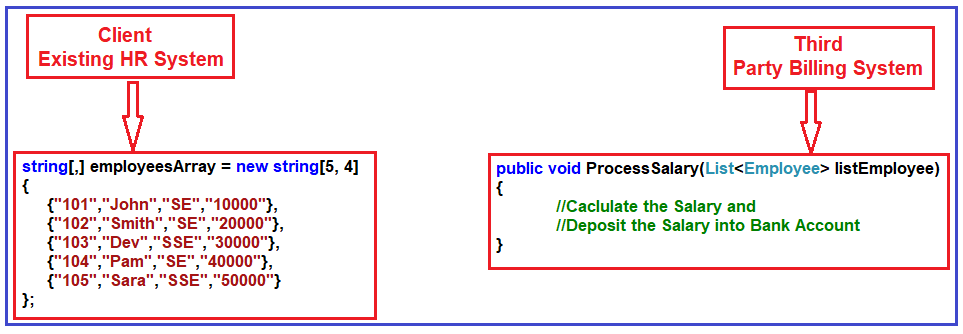
**What is the Adapter Design Pattern?**

The Adapter Design Pattern is a structural pattern that allows objects with incompatible interfaces to work together. It acts as a bridge between two incompatible interfaces. This pattern is useful when you want to use existing classes, but their interfaces do not match the one you need.

The Adapter Design Pattern acts as a bridge between two incompatible objects. Let’s say the first object is A and the second object is B. Object A wants to consume some of object B’s services. However, these two objects are incompatible and cannot communicate directly. In this case, the Adapter will come into the picture and act as a middleman or bridge between objects A and B. Now, object A will call the Adapter, and the Adapter will do the necessary transformations or conversions, and then it will call object B.

**Example to Understand Adapter Design Pattern in C#:**

Let us understand the Adapter Design Pattern with an Example, and then we will see the UML Diagram of the Adapter Design Pattern by comparing it with our Example. Please have a look at the following image. Here, you can see two interfaces, or you can say two systems. On the right-hand side, you can see the Third Party Billing System; on the left-hand side, you can see the Client, i.e., the Existing HR System. Now, we will see how these two systems are incompatible, and we will also see how we will make them compatible using Adapter Design Patterns in C#.

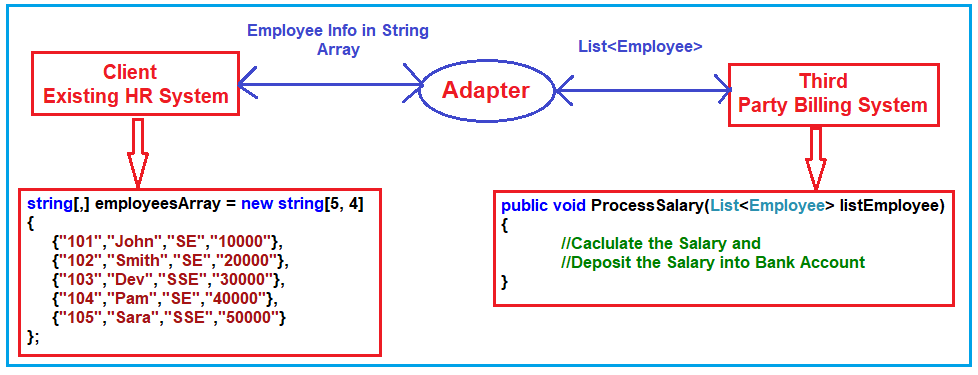


As you can see, the Third Party Billing System provides one functionality called ProcessSalary. What this ProcessSalary method will do is it will take the employee list (i.e., **List<Employee>**) as an input parameter and then loop through each employee, calculate the salary, and deposit the salary into the employee’s bank account.

On the left-hand side, i.e., in the Existing HR System, the employee information is stored as a string array. The HR System wants to process the salary of employees. Then, the HR System has to call the ProcessSalary method of the Third Party Billing System. But if you look at the HR system, the employee information is stored in a string array, and the ProcessSalary method of the Third Party Billing System wants data in List<Employee>. So, the HR System cannot call the Third Party Billing System directly because List<Employee> and string array are incompatible. So, these two systems are incompatible.

**How can we make these two incompatible systems work together?**

We can use the Adapter Design Pattern in C# to make these two systems or interfaces work together. We need to introduce an Adapter between the HR and Third Party Billing systems, as shown in the image below.



Now, the HR System will send the employee information as a String Array to the Adapter. Then, what this Adapter will do is it will read the employee information from the string array, populate the employee object, and then put each employee object into the List<Employee> collection. Then, the Adapter will send the List<Employee> to the ProcessSalary method of the Third Party Billing System. Then, the ProcessSalary method calculates each employee’s salary and deposits the salary into the Employee’s bank account.

So, in this way, we can make two incompatible interfaces work together with the help of the Adapter Design Pattern in C#. Again, the Adapter Design Pattern in C# can be implemented in two ways. They are as follows.

1. **Object Adapter Pattern**
2. **Class Adapter Pattern**

**Note:** We will discuss both the **Object Adapter Pattern and Class Adapter Pattern,** as well as we will also discuss the difference between them and when to use one over another.

**Implementation of Object Adapter Design Pattern in C#:**

Let us implement the example we discussed using the Object Adapter Design Pattern in C# step by step. Once we discuss the Example, we will see the UML Diagram of the Object Adapter Design Pattern by comparing it with our Examples. So, let us proceed and see how we can implement the Object Adapter Design Pattern in C#.

**Step 1: Creating Employee Class**

Create a class file named **Employee.cs** and copy and paste the following code. This class is going to be used by ThirdPartyBillingSystem (i.e., Adaptee) as well as by the Adapter. Here, we created the Employee with the required properties and then initialize the properties using the class constructor.

**namespace** *AdapterDesignPattern*

**{**

**public** **class** Employee

**{**

**public** **int** ID **{** **get**; **set**; **}**

**public** **string** Name **{** **get**; **set**; **}**

**public** **string** Designation **{** **get**; **set**; **}**

**public** **decimal** Salary **{** **get**; **set**; **}**

**public** Employee**(int** id, **string** name, **string** designation, **decimal** salary**)**

**{**

ID = id;

Name = name;

Designation = designation;

Salary = salary;

**}**

**}**

**}**

**Step2: Creating Adaptee**

This is going to be a class that contains the functionality that the client requires. However, this interface is not compatible with the client. So, create a class file named **ThirdPartyBillingSystem.cs** and copy and paste the following code. This class has the ProcessSalary method, which takes a list of employees as an input parameter and then processes each employee’s salary.

**using** *System;*

**using** *System.Collections.Generic;*

**namespace** *AdapterDesignPattern*

**{**

// The Adaptee contains some functionality that is required by the client.

// But this interface is not compatible with the client code.

**public** **class** ThirdPartyBillingSystem

**{**

//ThirdPartyBillingSystem accepts employee's information as a List to process each employee's salary

**public** **void** ProcessSalary**(**List**<**Employee**>** listEmployee**)**

**{**

**foreach** **(**Employee employee in listEmployee**)**

**{**

Console.WriteLine**(**"Rs." + employee.Salary + " Salary Credited to " + employee.Name + " Account"**)**;

**}**

**}**

**}**

**}**

**Step3: Creating ITarget interface**

This is going to be the domain-specific interface that is going to be used by the client. So, create an interface with the name **ITarget.cs** and then copy and paste the following code into it. This class defines the abstract ProcessCompanySalary method, which is going to be implemented by the Adapter. Again, the client will use this method to process the salary.

**namespace** *AdapterDesignPattern*

**{**

// The ITarget defines the domain-specific interface used by the client code.

// This interface needs to be implemented by the Adapter.

// The client can only see this interface i.e. the class which implements the ITarget interface.

**public** **interface** ITarget

**{**

**void** ProcessCompanySalary**(string[**,**]** employeesArray**)**;

**}**

**}**

**Step4: Create an Adapter**

This will be the class that implements the ITarget interface and has a reference to the Adaptee (**ThirdPartyBillingSystem**) object as we are using the Object Adapter Design Pattern. This class is responsible for communication between the Client and the Adaptee.

So, create a class file named **EmployeeAdapter.cs** and copy and paste the following code. This class implements the ITarget interface and provides the implementation for the **ProcessCompanySalary** method. This class also has a reference to the ThirdPartyBillingSystem (Adaptee) object. The ProcessCompanySalary method receives the employee information as a string array, converts the string array to a list of Employees, and then calls the ProcessSalary method on the ThirdPartyBillingSystem (Adaptee) object by providing the list of employees as an argument.

**using** *System;*

**using** *System.Collections.Generic;*

**namespace** *AdapterDesignPattern*

**{**

// This is the class that makes two incompatible interfaces or systems work together.

// The Adapter makes the Adaptee's interface compatible with the Target's interface.

**public** **class** EmployeeAdapter : ITarget

**{**

//To use Object Adapter Design Pattern, we need to create an object of ThirdPartyBillingSystem

ThirdPartyBillingSystem thirdPartyBillingSystem = new ThirdPartyBillingSystem**()**;

//The following will accept the employees in the form of string array

//Then convert the employee string array to List of Employees

//After conversation, it will call the Adaptee's Method to Process the Salaries

**public** **void** ProcessCompanySalary**(string[**,**]** employeesArray**)**

**{**

**string** Id = **null**;

**string** Name = **null**;

**string** Designation = **null**;

**string** Salary = **null**;

List**<**Employee**>** listEmployee = new List**<**Employee**>()**;

**for** **(int** i = 0; i **<** employeesArray.GetLength**(**0**)**; i++**)**

**{**

**for** **(int** j = 0; j **<** employeesArray.GetLength**(**1**)**; j++**)**

**{**

**if** **(**j == 0**)**

**{**

Id = employeesArray**[**i, j**]**;

**}**

**else** **if** **(**j == 1**)**

**{**

Name = employeesArray**[**i, j**]**;

**}**

**else** **if** **(**j == 2**)**

**{**

Designation = employeesArray**[**i, j**]**;

**}**

**else**

**{**

Salary = employeesArray**[**i, j**]**;

**}**

**}**

listEmployee.Add**(**new Employee**(**Convert.ToInt32**(**Id**)**, Name, Designation, Convert.ToDecimal**(**Salary**)))**;

**}**

Console.WriteLine**(**"Adapter converted Array of Employee to List of Employee"**)**;

Console.WriteLine**(**"Then delegate to the ThirdPartyBillingSystem for processing the employee salary\n"**)**;

thirdPartyBillingSystem.ProcessSalary**(**listEmployee**)**;

**}**

**}**

**}**

**Step5: Client**

Here, the client will be our HR System (i.e., the Main method of the Program class). Please modify the Main method as shown below. Notice that we have the employee information as a string array here. Then, we create an instance of **EmployeeAdapter** and call the **ProcessCompanySalary** method by passing the string array as an argument. So, with the help of the Adapter (i.e., EmployeeAdapter object), the Client and the Third Party Billing System now work together.

**using** *System;*

**namespace** *AdapterDesignPattern*

**{**

//Client

//The Client is Incompatible with ThirdPartyBillingSystem

**class** Program

**{**

**static** **void** Main**(string[]** args**)**

**{**

//Storing the Employees Data in a String Array

**string[**,**]** employeesArray = new **string[**5, 4**]**

**{**

**{**"101","John","SE","10000"**}**,

**{**"102","Smith","SE","20000"**}**,

**{**"103","Dev","SSE","30000"**}**,

**{**"104","Pam","SE","40000"**}**,

**{**"105","Sara","SSE","50000"**}**

**}**;

//The EmployeeAdapter Makes it possible to work with Two Incompatible Interfaces

Console.WriteLine**(**"HR system passes employee string array to Adapter\n"**)**;

ITarget target = new EmployeeAdapter**()**;

target.ProcessCompanySalary**(**employeesArray**)**;

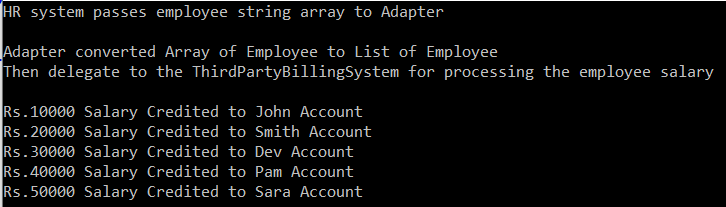
Console.Read**()**;

**}**

**}**

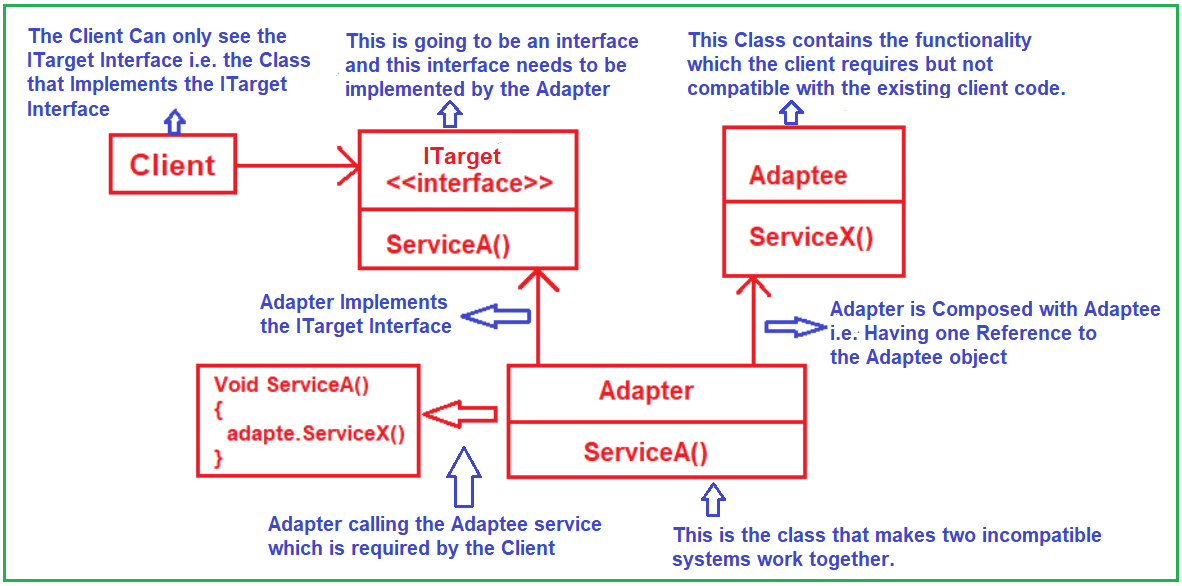
**}**

**Output:**



**UML Diagram of Object Adapter Design Pattern in C#:**

Let us understand the UML Diagram of the Object Adapter Design Pattern by comparing the components with our example. To understand the Class or UML diagram and the different components of the Object Adapter Design Pattern, please look at the following diagram. As you can see, the client uses the ITarget Interface and creates an instance of the Adapter, and using the Adapter instance, the client communicates with the Adaptee. The Adapter is the component that makes it possible to work with two different incompatible interfaces.



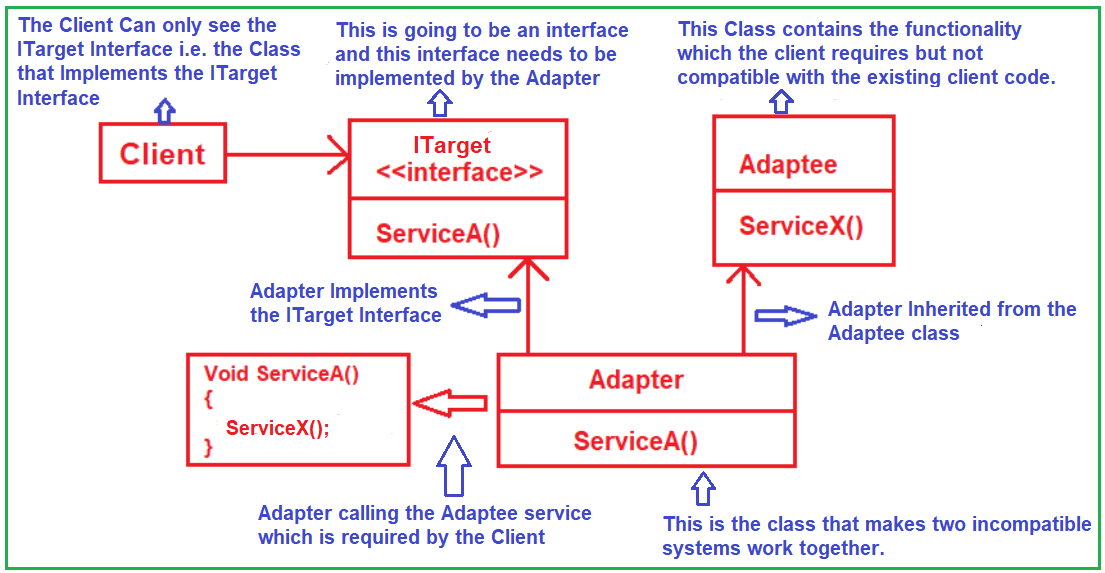
The Adapter Design Pattern is composed of four components. They are as follows:

1. **Client:** The Client class can only see the ITarget interface, i.e., the class that implements the ITarget interface, i.e., the Adapter (in our example, it is the EmployeeAdapter). Using that Adapter (EmployeeAdapter) object, the client will communicate with the Adaptee, which is incompatible with the client.
2. **ITarget:** This is going to be an interface that needs to be implemented by the Adapter. The client can only see this interface, i.e., the class which implements this interface.
3. **Adapter:** This class makes two incompatible interfaces or systems work together. The Adapter class implements the ITrager interface and provides the implementation for the interface method. This class is also composed of the Adaptee, i.e., it has a reference to the Adaptee object as we are using the Object Adapter Design Pattern. In our example, the EmployeeAdapter class implements the ITarget Interface and provides implementations to the ProcessCompanySalary method of the ITarget Interface. This class also has a reference to the ThirdPartyBillingSystem object.
4. **Adaptee:** This class contains the client’s required functionality but is incompatible with the existing client code. So, it requires some adaptation or transformation before the client can use it. It means the client will call the Adapter, and the Adapter will do the required conversions and then make a call to the Adaptee.

This is all about the Object Adapter Design Pattern in C#. Let us proceed and see how to achieve the same using the Class Adapter Design Pattern in C#.

**Understanding Class Adapter Design Pattern in C#:**

This is another approach to implementing the Adapter Design Pattern in C#. In this approach, the Adapter calls will implement the ITarget interface and inherit from the Adaptee class. That means the Adapter class will now be a child of the Adaptee class. So, instead of creating a reference variable of Adaptee to call the Adaptee method, it can call that method directly as it is available via inheritance. Before implementing the same example using the Class Adapter Design Pattern, let us first understand the class diagram of the Class Adapter Design Pattern. Please have a look at the following image.



The class diagram is identical to the Object Adapter Design Pattern class diagram. The only difference is that the Adapter class now implements the Target interface and is inherited from the Adaptee class. In the case of the Object Adapter Design Pattern, the adapter has a reference to the Adaptee object, and using that reference, it will call the adaptee methods. But in the case of the Class Adapter Design Pattern, the adapter will call the inherited method of the Adaptee class directly.

**Implementation of Class Adapter Design Pattern in C#:**

Let us implement the previous example using the Class Adapter Design Pattern in C# step by step. The implementation is exactly the same as the Object Adapter Design Pattern Implementation. The only difference is in the EmployeeAdapter class. Now, in the EmployeeAdapter class, instead of Creating a reference variable of the ThirdPartyBillingSystem class, we will make this EmployeeAdapter class inherit from the ThirdPartyBillingSystem. After the required transformation or conversations, we need to call the ProcessSalary method.

So, please modify the EmployeeAdapter class as shown below to use the Class Adapter Design Pattern in C#. The EmployeeAdapter class is now inherited from the Adaptee, i.e., the ThirdPartyBillingSystem class, and implements the ITarget interface.

**using** *System;*

**using** *System.Collections.Generic;*

**namespace** *AdapterDesignPattern*

**{**

// This is the class that makes two incompatible interfaces or systems work together.

// The Adapter makes the Adaptee's interface compatible with the Target's interface.

// To use Class Adapter Pattern, we need to inherit the Adapter class from the Adaptee class

**public** **class** EmployeeAdapter : ThirdPartyBillingSystem, ITarget

**{**

//The following will accept the employees in the form of string array

//Then convert the employee string array to List of Employees

//After conversation, it will call the Adaptee's Method to Process the Salaries

**public** **void** ProcessCompanySalary**(string[**,**]** employeesArray**)**

**{**

**string** Id = **null**;

**string** Name = **null**;

**string** Designation = **null**;

**string** Salary = **null**;

List**<**Employee**>** listEmployee = new List**<**Employee**>()**;

**for** **(int** i = 0; i **<** employeesArray.GetLength**(**0**)**; i++**)**

**{**

**for** **(int** j = 0; j **<** employeesArray.GetLength**(**1**)**; j++**)**

**{**

**if** **(**j == 0**)**

**{**

Id = employeesArray**[**i, j**]**;

**}**

**else** **if** **(**j == 1**)**

**{**

Name = employeesArray**[**i, j**]**;

**}**

**else** **if** **(**j == 2**)**

**{**

Designation = employeesArray**[**i, j**]**;

**}**

**else**

**{**

Salary = employeesArray**[**i, j**]**;

**}**

**}**

listEmployee.Add**(**new Employee**(**Convert.ToInt32**(**Id**)**, Name, Designation, Convert.ToDecimal**(**Salary**)))**;

**}**

Console.WriteLine**(**"Adapter converted Array of Employee to List of Employee"**)**;

Console.WriteLine**(**"Then delegate to the ThirdPartyBillingSystem for processing the employee salary\n"**)**;

//Call the Base Class ProcessSalary Method to Process the Salary

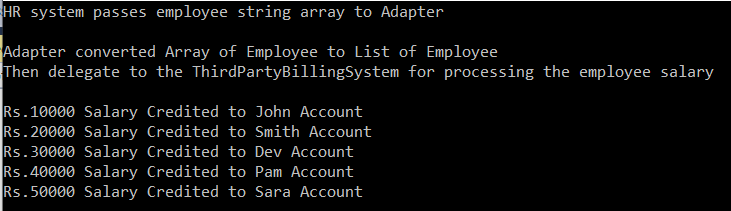
ProcessSalary**(**listEmployee**)**;

**}**

**}**

**}**

**Output:**



**When to use the Object Adapter Pattern and Class Adapter Pattern in C#?**

It is completely based on the situation. For example, if you have a Java class and want to make it compatible with the dot net class, then you need to use the Object Adapter Design Pattern because it is not possible to make the inheritance. On the other hand, if both the classes are within the same project and using the same programming language, and if inheritance is possible. then you need to go for the Class Adapter Design Pattern.

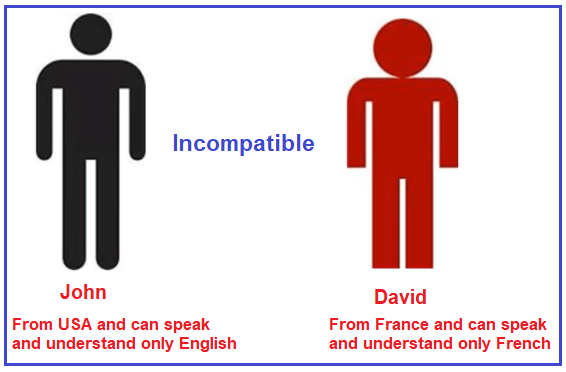
**When to use the Adapter Design Pattern in Real-Time Applications?**

The Adapter Design Pattern in C# is particularly useful in the following scenarios:

* **Integration with Third-party or Legacy Systems:** When your application needs to interact with an external system or a legacy system, and the interfaces of the external systems are not compatible with your application’s interfaces.
* **Reusing Existing Code:** If you have existing classes with functionality that you need to use, but their interfaces don’t match the ones your system currently uses, an adapter can bridge this gap.
* **Creating a Common Interface for Different Classes:** When you have several classes with different interfaces but want to treat them uniformly through a common interface.
* **Supporting Multiple Data Sources:** When your application needs to handle data from different sources (like databases, file systems, web services) but wants to process them in a uniform manner.
* **Testing and Mocking:** Adapters can be used to create stubs or mocks for unit testing, especially when the actual objects are cumbersome to use in a test environment (like database connections or external services).
* **Providing Backward Compatibility:** When updating an application or library, adapters can be used to maintain backward compatibility with the old versions of APIs or data models.
* **Cross-Platform Compatibility:** In scenarios where you need to provide support for different platforms or environments while keeping the rest of the application code consistent.

**Adapter Design Pattern Real-Time Example in C# – Language Translator**

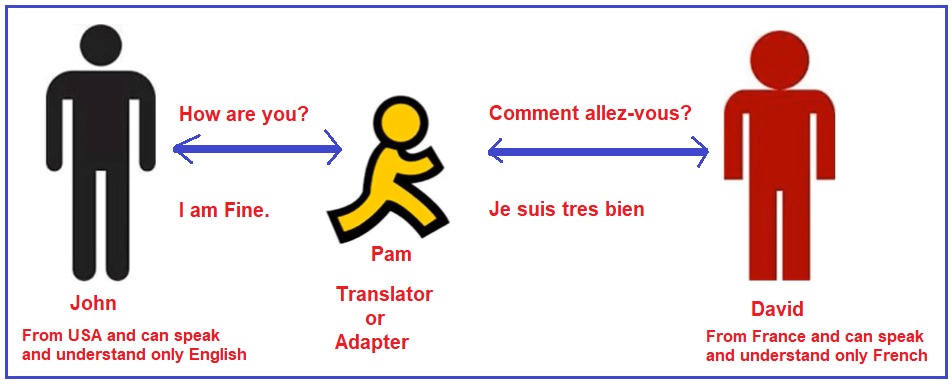
Let us understand the Language Translator Example in C# using Adapter Design Pattern. Please have a look at the following diagram for a better understanding. On the left-hand side, you can see a person called John and on the right-hand side, you can see a person called David.



The person John is from the USA and he can only speak and understand only English. On the other hand, person David is from France and he can only speak and understand French. Now, John wants to speak something to David and David wants to speak something back to John. But, currently, it is not possible because John only knows English and he can’t understand French. In the same way, David knows only French and he can’t understand English. So the above two interfaces are incompatible as they can’t communicate.

**How to Make Them Compatible?**

In order to make them compatible with each other, what we need to do is, we need to introduce a Translator  (Middleman) between John and David which is shown in the below image. Here, the person Pam acts as a Translator. Pam knows both English and French. She can speak and understand English, as well as translate English to French. She also speaks and understands French, as well as translates French into English. So, here, Pam is nothing but an Adapter.



Suppose John wants to ask **how are you** to David. Then what John will do is, he will ask how are you to Pam, and Pam will convert how are you to French and ask David. David will reply in French to Pam and Pam again converts that French to English and return to John. So, in this case, both John and David will communicate with each other. Here, the important thing is Pam and Pam is acting as an Adapter between John and David.

**Implementation of Adapter Design Pattern Real-Time Example using C#:**

Let us implement the above-discussed Language Translator Example step by step in C# using Adapter Design Pattern. As we already discussed in our previous article, the Adapter Design Pattern involves four components (Target, Client, Adaptee, and Adapter). Let us create these components by comparing them with our example.

**Step1: Creating EnglishSpeaker Interface (Adaptee)**

Create an interface with the name IEnglishSpeaker and then copy and paste the following code into it. This interface is having two methods i.e. AskQuestion and AnswerFortheQuestion and both these methods expect parameter words.

**namespace** *AdapterDesignPatternRealTimeExample*

**{**

**public** **interface** IEnglishSpeaker

**{**

**string** AskQuestion**(string** Words**)**;

**string** AnswerFortheQuestion**(string** Words**)**;

**}**

**}**

In our example, John is a person who can speak and understand only English. So create a class file with the name **John.cs** and then copy and paste the following code into it. This John class implements the IEnglishSpeaker interface and provides implementations for the two abstract methods i.e. AskQuestion and AnswerFortheQuestion.

**using** *System;*

**namespace** *AdapterDesignPatternRealTimeExample*

**{**

// John is from USA, So he can speak and understand only English

**public** **class** John : IEnglishSpeaker

**{**

**public** **string** AskQuestion**(string** Words**)**

**{**

Console.WriteLine**(**"Question Asked by John [English Speaker and Can understand only English] : " + Words**)**;

ITarget target = new Pam**()**;

**string** replyFromDavid = target.TranslateAndTellToOtherPerson**(**Words, "French"**)**;

**return** replyFromDavid;

**}**

**public** **string** AnswerFortheQuestion**(string** Words**)**

**{**

**string** reply = **null**;

**if** **(**Words.Equals**(**"where are you?", StringComparison.InvariantCultureIgnoreCase**))**

**{**

reply = "I am in USA";

**}**

**return** reply;

**}**

**}**

**}**

**Step2: Creating FrenchSpeaker Interface (Adaptee)**

Create an interface with the name IFrenchSpeaker and then copy and paste the following code into it. This interface is also having two methods (AskQuestion and AnswerFortheQuestion) and again these two methods expect parameter words.

**namespace** *AdapterDesignPatternRealTimeExample*

**{**

**public** **interface** IFrenchSpeaker

**{**

**string** AskQuestion**(string** Words**)**;

**string** AnswerFortheQuestion**(string** Words**)**;

**}**

**}**

In our example, David is the person who can only speak and understand only French. So, create a class with the name David and then copy and paste the following code into it. This class is going to implement the IFrenchSpeaker interface and provide implementations for the AskQuestion and AnswerFortheQuestion abstract methods.

**using** *System;*

**namespace** *AdapterDesignPatternRealTimeExample*

**{**

// David is from France and can speak and understand only French

**public** **class** David : IFrenchSpeaker

**{**

**public** **string** AskQuestion**(string** Words**)**

**{**

Console.WriteLine**(**"Question Asked by David [French Speaker and Can understand only French] : " + Words**)**;

ITarget target = new Pam**()**;

**string** replyFromJohn = target.TranslateAndTellToOtherPerson**(**Words, "English"**)**;

**return** replyFromJohn;

**}**

**public** **string** AnswerFortheQuestion**(string** Words**)**

**{**

**string** reply = **null**;

**if** **(**Words.Equals**(**"comment allez-vous?", StringComparison.InvariantCultureIgnoreCase**))**

**{**

reply = "Je suis très bien";

**}**

**return** reply;

**}**

**}**

**}**

**Step3: Creating Target interface**

Create an interface with the name ITarget and then copy and paste the following code into it. The ITarget interface defines the abstract TranslateAndTellToOtherPerson method which is going to be implemented by the Adapter.

**namespace** *AdapterDesignPatternRealTimeExample*

**{**

**public** **interface** ITarget

**{**

**string** TranslateAndTellToOtherPerson**(string** words, **string** convertToWhichLanguage**)**;

**}**

**}**

**Step4: Creating Pam (Translator or Adapter)**

Create a class with the name Pam and then copy and paste the following code into it. The following Pam class implements the ITarget interface and provides implementations for the TranslateAndTellToOtherPerson method. What this TranslateAndTellToOtherPerson method does is, it will translate the word to English and talk to John and again it also translates the word to French and talks to David. This method acts as a two-way communicator (it can communicate with both persons).

This class is also having two methods. One method is used to convert English to French and the other method is used to convert French to English.

**using** *System;*

**using** *System.Collections.Generic;*

**namespace** *AdapterDesignPatternRealTimeExample*

**{**

// Adapter or Translator

// Pam can speak and understand both English and French

// She acts as an Adapter or Translator

**public** **class** Pam:ITarget

**{**

**static** Dictionary **<** **string**, **string** **>** EnglishFrenchDictionary = new Dictionary **<** **string**, **string** **>** **()**;

**static** Dictionary **<** **string**, **string** **>** FrenchEnglishDictionary = new Dictionary **<** **string**, **string** **>** **()**;

David david = new David **()**;

John john = new John **()**;

**static** Pam **()**

**{**

EnglishFrenchDictionary.Add **(**"how are you?", "comment allez-vous?"**)**;

EnglishFrenchDictionary.Add **(**"I am in USA", "Je suis aux Etats-Unis"**)**;

FrenchEnglishDictionary.Add **(**"Je suis trC(s bien", "I am fine"**)**;

FrenchEnglishDictionary.Add **(**"oC9 C\*tes-vous?", "where are you?"**)**;

**}**

**public** **string** TranslateAndTellToOtherPerson **(string** Words, **string** ConvertToWhichLanguage**)**

**{**

**if** **(**ConvertToWhichLanguage.Equals **(**"English", StringComparison.InvariantCultureIgnoreCase**))**

**{**

**string** EnglishWords = ConvertToEnglish **(**Words**)**;

Console.WriteLine **(**"\nPam Converted \"" + Words + " \" to \"" + EnglishWords + " and send the question to John"**)**;

**string** EnglishWordsReply = john.AnswerFortheQuestion **(**EnglishWords**)**;

Console.WriteLine **(**"Pam Got reply from John in English : " + "\"" + EnglishWordsReply + "\""**)**;

**string** FrenchConverted = ConvertToFrench **(**EnglishWordsReply**)**;

Console.WriteLine **(**"Pam Converted " + "\"" + EnglishWordsReply + "\"" + " to " + "\"" + FrenchConverted + "\"" + " and send back to David\n"**)**;

**return** FrenchConverted;

**}**

**else** **if** **(**ConvertToWhichLanguage.Equals**(**"French", StringComparison.InvariantCultureIgnoreCase**))**

**{**

**string** FrenchWords = ConvertToFrench **(**Words**)**;

Console.WriteLine **(**"\nPam Converted \"" + Words + " \" to \"" + FrenchWords + " and send the question to David"**)**;

**string** FrenchWordsReply = david.AnswerFortheQuestion **(**FrenchWords**)**;

Console.WriteLine **(**"Pam Got reply from David in French : " + "\"" +FrenchWordsReply + "\""**)**;

**string** EnglishConverted = ConvertToEnglish **(**FrenchWordsReply**)**;

Console.WriteLine **(**"Pam Converted " + "\"" + FrenchWordsReply + "\"" + " to " + "\"" + EnglishConverted + "\"" +" and send back to John\n"**)**;

**return** EnglishConverted;

**}**

**else**

**{**

**return** "Sorry Cannot Covert";

**}**

**}**

**public** **string** ConvertToFrench **(string** Words**)**

**{**

**return** EnglishFrenchDictionary**[**Words**]**;

**}**

**public** **string** ConvertToEnglish **(string** Words**)**

**{**

**return** FrenchEnglishDictionary**[**Words**]**;

**}**

**}**

**}**

**Note:** This class acts as an Adapter as it is adapting something before communicating.

**Step5: Client**

Our main method is going to be the client for our application. So, please modify the Main method as shown below.

**using** *System;*

**namespace** *AdapterDesignPatternRealTimeExample*

**{**

**class** Program

**{**

**static** **void** Main**(string[]** args**)**

**{**

**string** replyFromDavid = new John**()**.AskQuestion**(**"how are you?"**)**;

Console.WriteLine**(**"Reply From David [French Speaker can Speak and Understand only French] : " + replyFromDavid**)**;

Console.WriteLine**()**;

**string** replyFromJohn = new David**()**.AskQuestion**(**"où êtes-vous?"**)**;

Console.WriteLine**(**"Reply From John [English Speaker can Speak and Understand only English] : " + replyFromJohn**)**;

Console.Read**()**;

**}**

**}**

**}**

**Output:**

