[**https://dotnettutorials.net/lesson/adapter-design-pattern/**](https://dotnettutorials.net/lesson/adapter-design-pattern/)

**Adapter Design Pattern in C# with Examples**

The Adapter Design Pattern falls under the category of Structural Design Pattern.

**What is the Adapter Design Pattern?**

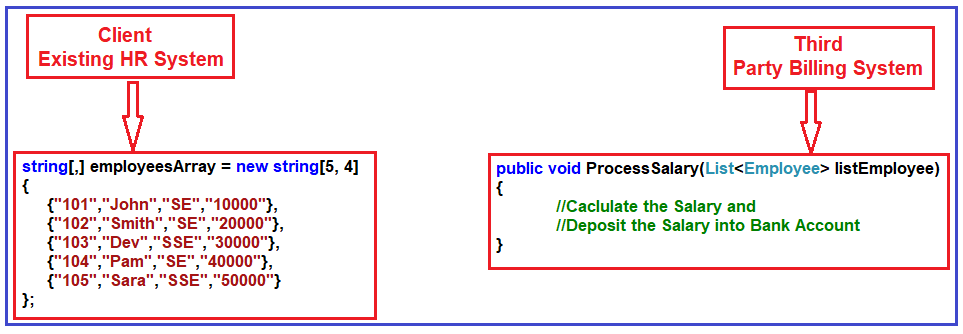
1. **Understanding the Object and Class Adapter Pattern.**
2. **Implementing both Class and Object Adapter Pattern using C#.**
3. **When to use the Adapter Design Pattern in C#?**
4. **When to use the Object Adapter pattern and when to use the Class Adapter Pattern?**

**What is the Adapter Design Pattern?**

The Adapter Design Pattern works as a bridge between two incompatible interfaces. This design pattern involves a single class called adapter which is responsible for communication between two independent or incompatible interfaces. So, in simple words, we can say that the Adapter Pattern helps two incompatible interfaces to work together. If this is not clear at the moment then don’t worry we will understand this with an example.

**Understanding the Adapter Design Pattern in C# using an Example:**

Please have a look at the following image. On the right-hand side, you can see the Third Party Billing System and on the left 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#.

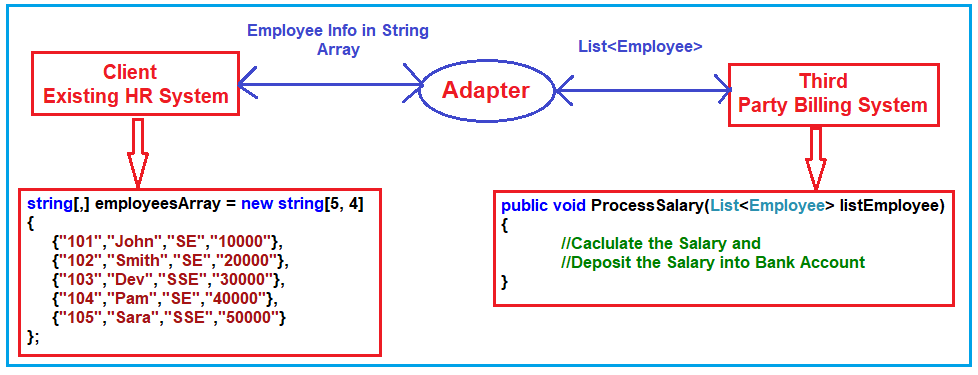


As you can see, the Third Party Billing System provides one functionality called ProcessSalary. What this ProcessSalary method will do is, it will take the employee list (i.e. **List<Employee>**) as an input parameter and then loop through each employee and calculate the salary and deposit the salary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

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

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

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



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

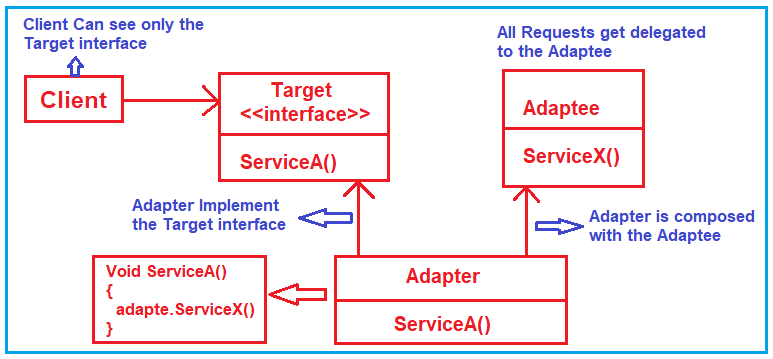
So, in this way, we can make two incompatible interfaces work together with the help of the Adapter Design Pattern. Again the Adapter Design Pattern can be implemented in two ways. They are as follows.

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

**Note:** In this article, we will discuss the Object Adapter Pattern and in the next article, we will discuss the Class Adapter Pattern as well as we will also discuss the difference between them and when to use one over another.

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

An Object Adapter delegates to an adaptee object. Let us understand the class and diagram first. In order to understand the class diagram and the different components involved in the Adapter Design Pattern please have a look at the following diagram.



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

1. **Client:** The Client class can only see the Target interface.
2. **Target:** It an interface and this interface needs to be implemented by the Adapter and the client can see only this interface.
3. **Adapter:** This is a class that makes two incompatible systems work together. The Adapter class implements the Trager interface and provides the implementation for the ServiceA method. This class is also composed of the Adaptee i.e. it has a reference to the Adaptee object.
4. **Adaptee:** This class contains the functionality which the client requires but it’s not compatible with the existing client code. So, it requires some adaptation before the client code can use it. It means the client will call the Adapter and the Adapter will do the conversion if required and then it will make a call to the Adaptee.

This is what object adapter design pattern.

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

Let us implement the example that we discussed using Object Adapter Design Pattern in C# step by step.

**Step1: Creating Employee class**

Create a class file with the name **Employee.cs** and then copy and paste the following code in it. This class is going to be used by Third Party Billing System (i.e. Adaptee) as we 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**

Create a class file with the name ThirdPartyBillingSystem.cs and then copy and paste the following code in it. This class is having the ProcessSalary method which takes a list of employees as an input parameter and then processes the salary of each employee.

**using** *System;*

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

**namespace** *AdapterDesignPattern*

**{**

**public** **class** ThirdPartyBillingSystem

**{**

//ThirdPartyBillingSystem accepts employees information as a List to process each employee 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 Target interface**

Create an interface with the name **ITarget** and then copy and paste the following code in it. This class defines the abstract ProcessCompanySalary method which is going to be implemented by the Adapter. Again the client is going to use this method to process the salary.

**namespace** *AdapterDesignPattern*

**{**

**public** **interface** ITarget

**{**

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

**}**

**}**

**Step4: Creating Adapter**

Create a class file with the name EmployeeAdapter.cs and then copy and paste the following code in it. This class implements the ITarget interface and provides the implementation for the ProcessCompanySalary method. This class also has a reference to the ThirdPartyBillingSystem object. The ProcessCompanySalary method receives the employee information as a string array and then converts the string array to a list of employees and then calls the ProcessSalary method on the ThirdPartyBillingSystem object bypassing the list of employees as an argument.

**using** *System;*

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

**namespace** *AdapterDesignPattern*

**{**

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

**{**

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

**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 == 1**)**

**{**

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 is going to be our HR System. Please modify the Main method as shown below. Notice, here we have the employee information in the form of the string array. Then we create an instance of EmployeeAdapter and call the ProcessCompanySalary method by passing the string array as an argument.

**namespace** *AdapterDesignPattern*

**{**

**class** Program

**{**

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

**{**

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"**}**

**}**;

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

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

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

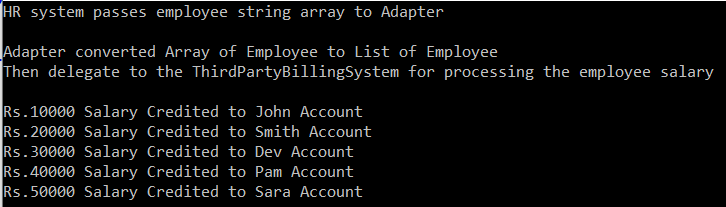
Console.Read**()**;

**}**

**}**

**}**

**Output:**



This is all about the object adapter design pattern. Let’s see how to achieve the same thing using the class adapter design pattern.