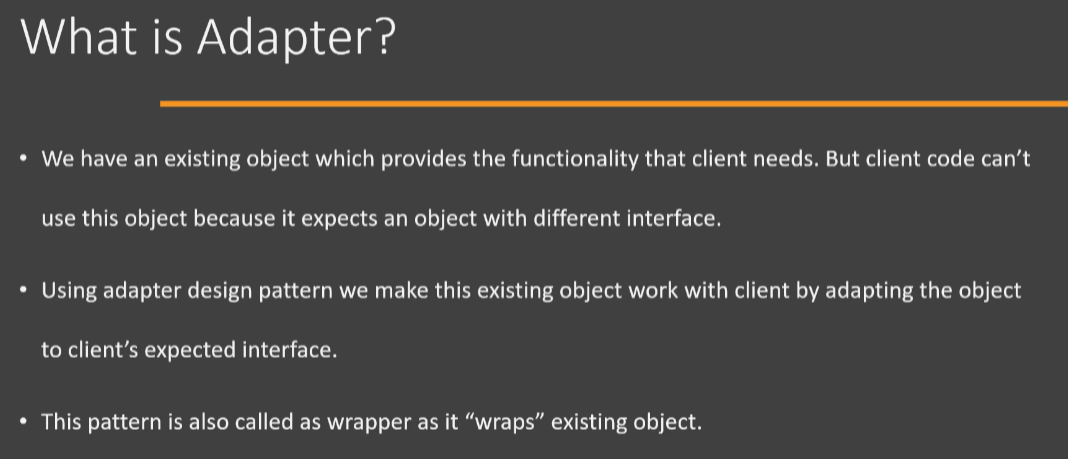
Adapter Design Pattern

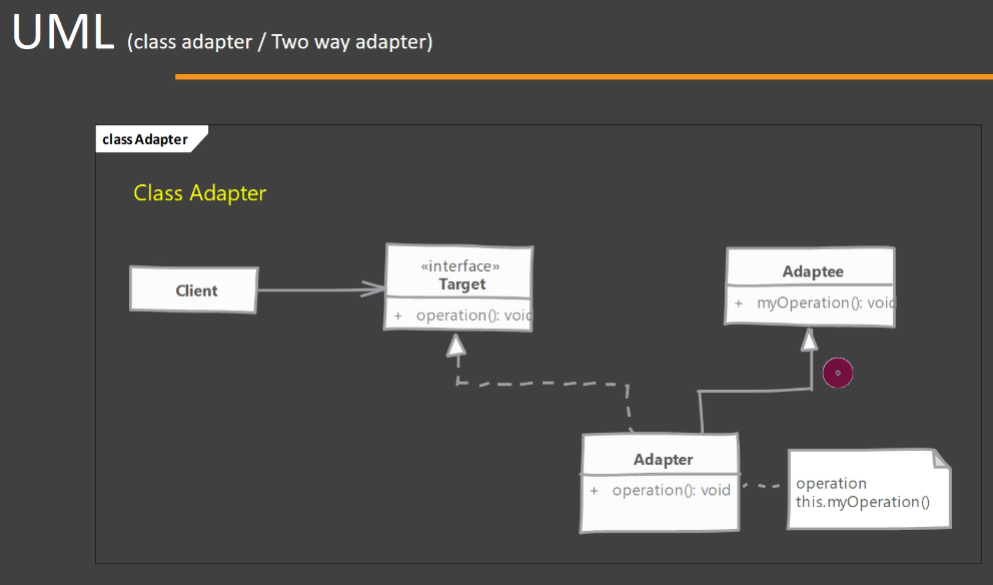
1. Adapter – Introduction:

Now we are going to learn about Adapter design pattern. So this is a structural design pattern and we are going to see why we need an Adapter design pattern in first place. So let's say you have an object in your application, so you have a class which has implemented some functionality and you can create object of that particular class. And there is a different part of your code maybe written by somebody else who can use or who needs that particular functionality which is provided by our existing class, but let's say that particular code expects an object which is implementing a completely different interface. So now you have a problem because you have actually an object which can provide that functionality. The only problem here is that the client code is expecting the object to be of a different interface. And this is the problem which Adapter design pattern tries to solve. Using Adapter, we can make this existing object work with client by adapting the object to whatever interface is expected by our client code. Now, this pattern is also called as wrapper as it wraps our existing object.

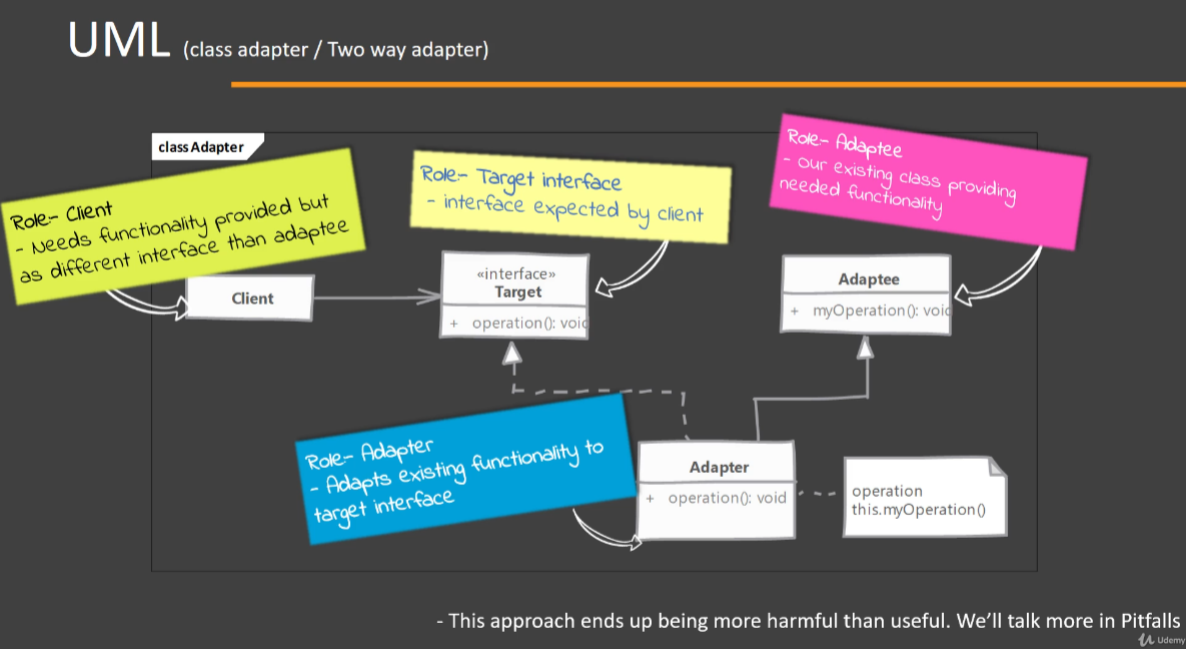


Now we'll understand better how this whole problem is and how an Adapter works by looking at the UML diagram. All right, so here we have a UML diagram of Adapter design pattern. Now, one point that you should remember is that there are two variations of this design pattern. One is called as a 'Class Adapter', which is also called as 'Two Way Adapter'. And there is also what is called an 'object adapter'. We're going to look at both of these one after another.

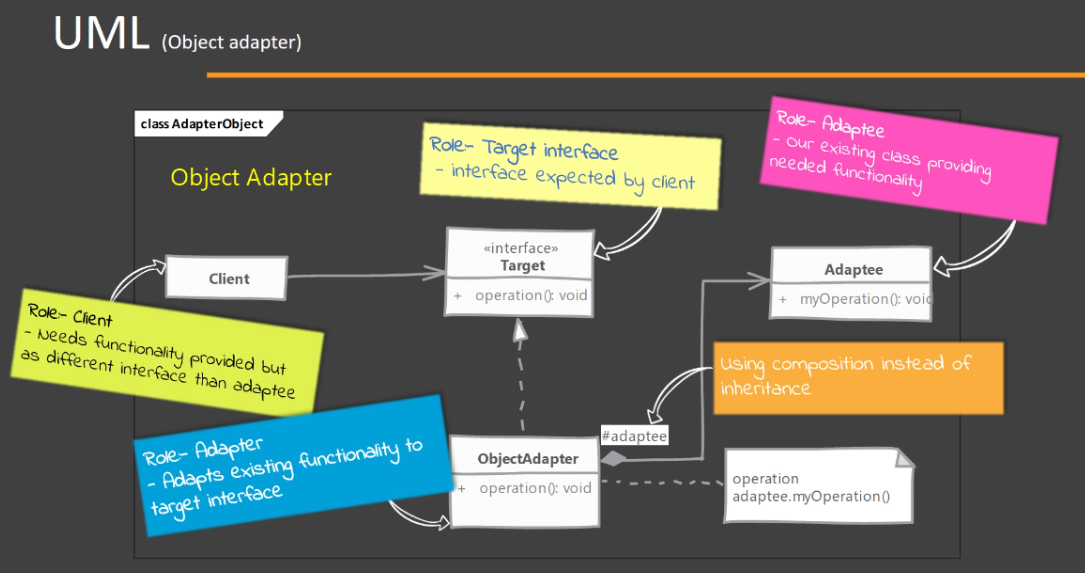
So to begin with, here we have a Class Adapter. Adaptee is the class which provides the functionality which is needed by the Client. So it has a method called as myOperation() that does exactly the thing that client needs. However, the Client is expecting an object which is implementing Target interface, and this interface has a method called as operation(). So even though we have the functionality that is needed by Client in the Adaptee class, it is not implementing this particular interface(i.e., Target) which is expected by Client. And to solve this problem we use Adapter class. So Adapter, in 'Class Adapter', is a class which extends from our existing class(i.e, Adaptee) which provides the functionality, and it implements this Target interface which is expected by our Client. So in the implementation of our operation() method, which is specified by the Target interface, we simply call this.myOperation(). Since our Adapter class already extends from the class(i.e., Adaptee) which provides this functionality, the myOperation() method code is already available here(i.e., inside Adapter class) and in the overridden implemented method[i.e., operation()], we simply delegate to that existing method[i.e., myOperation()]. All right, so this is how the Class Adapter design pattern works.



Now let's look at these roles once more so that you will get a better understanding of what this these roles do. So we have Adaptee which is an existing class that has already provided the functionality that is needed by Client. Now Client needs this functionality or can make use of that functionality, but it expects a completely different interface than Adaptee, and that interface is Target. So our Adapter adapts our existing class to the Client by implementing this interface and extending from Adaptee. Now this approach ends up being more harmful than useful, and we are going to talk about the problems of using ‘Class Adapter’ when we are discussing pitfalls of this design pattern.



Now I mentioned there is another way that we can implement this design pattern and that is using the 'Object Adapter'. And this is the preferred way that you should implement the Adapter design pattern. So here again, we have similar situation. We have an Adaptee class which is our existing class that provides the functionality needed by Client. We have Target interface which is expected by our Client. And lastly, we have an ObjectAdapter. Now here, instead of extending from our existing class, what we do is we simply implement the Target interface and we have an inner object, we have an object of Adaptee inside this ObjectAdapter. So we are making use of composition instead of inheritance to provide the functionality of Adaptee. So while implementing this operation() method inside ObjectAdapter, what we do is that we use our object(i.e., instance of Adaptee class) which is contained inside ObjectAdapter and call the myOperation() method on that object to provide the functionality. And that's why this is called as an Object Adapter because it adapts an object instead of extending from our class. So this is the preferred way to implement Adapter design pattern.



1. Adapter – Implementation Steps:

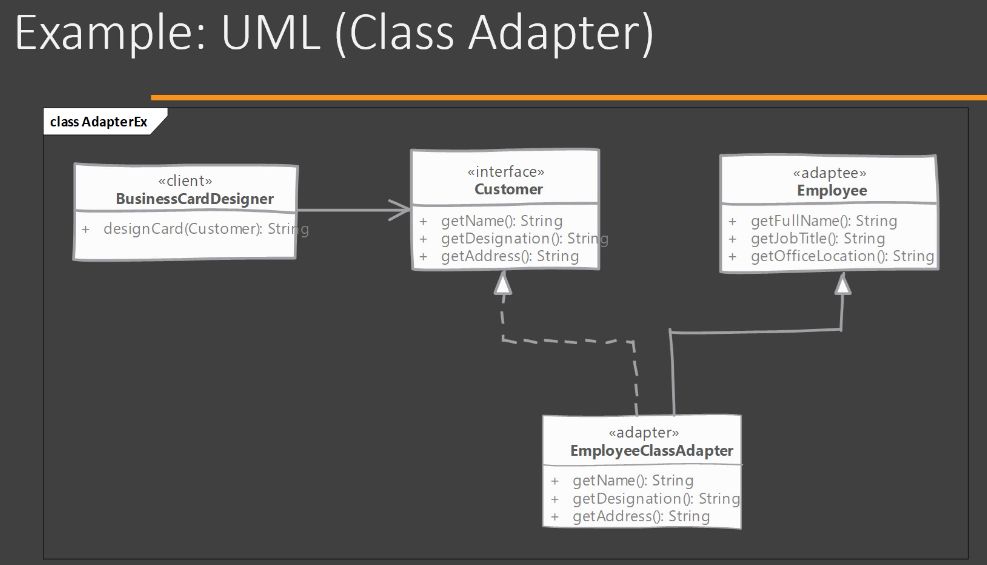
Now let's look at the steps that we need to follow in order to implement Adapter design pattern in Java. So we of course start by creating the Adapter class. Now, this Adapter class must implement the Target interface. It doesn't matter if you are trying out 'Class Adapter' or 'Object Adapter', your Adapter class must implement the interface(i.e., Target) which is expected by the Client. Now first we are going to try out a 'Class Adapter' and to do that we are simply going to extend from our existing class(i.e., Adaptee) in the 'Class Adapter' implementation. We are simply going to forward the method call that we get to another method which is inherited from the Adaptee class. So in the UML you saw that we already have access to our existing functionality that is provided by Adaptee class, while implementing the Target interface, in those method implementations[i.e., operation()], we are simply going to delegate this method call. Next, we are going to try out the Object Adapter here. Again, we need to implement the Target interface, but we are going to have/accept an object of the Adaptee or our existing class in the constructor of Adapter class so that we are going to use the composition instead of inheritance. And in the method implementations of the interface Target interface, we are going to delegate that call to this particular object and call myOperation() method on this particular object. Now, in Object Adapter, we should take the Adaptee or the Adaptee object as an argument in our constructor. That is the preferred way to do the Adapter implementation. There is another way that you can do. You can instantiate the Adaptee class object in the constructor or whenever you need. But the problem with that approach is that we tightly become coupled with the concrete implementation of the Adaptee. There is a benefit in accepting the concrete object in the constructor of Adapter class, we can create our Adapter object by using subclass of our Adaptee in future if we need, we can give the object of subclass of Adaptee class. So these are the steps that you need in order to implement an Adapter.

A screenshot of a computer program

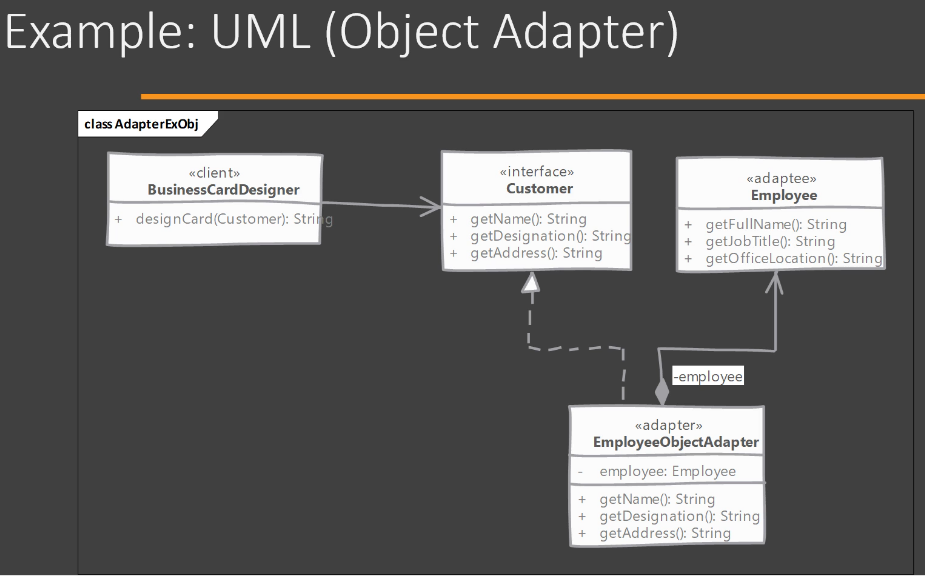
Description automatically generated

1. Adapter – Hands-on Example UML:

We are going to try out both the 'Class Adapter' and the 'Object Adapter', so we are going to look at two UML diagrams which basically implement the same problem. But first we are going to implement our Adapter as a 'Class Adapter' or 'Two Way Adapter' and then we are going to implement the 'Object Adapter'. So here we have a class called as BusinessCardDesigner, and this is the client code which expects an object. It has a method called as designCard(), which expects an object implementing the Customer interface. So here we have the Customer interface, which has some methods in it which are used by our client(i.e., BusinessCardDesigner). However, in our code we already have a class called as Employee. Now this class has all the information/operations that is needed by the BusinessCardDesigner. However, you can see that the Employee is not of type Customer but designCard() method of BusinessCardDesigner expects an object of type Customer. So we are going to create our class adapter called as EmployeeClassAdapter and we are going to extend from our Employee class, implement the Customer interface and provide implementation for these methods[getName(), getDesignation() and getAddress()].



Let's also take a look at the 'Object Adapter' UML diagram. In the 'Object Adapter', we have the same problem. We are going to solve the same problem. We have the BusinessCardDesigner class which expects Customer interface type of object and we have our existing Employee class. But here we are going to create a new class EmployeeObjectAdapter, which is going to only implement the Customer interface and we are going to accept the object of Employee class in our constructor of EmployeeObjectAdapter class and in the implementation of these methods[getName(), getDesignation() and getAddress()], we are going to delegate to our Employee object. So this is a quite a simple example and it's quite a simple design pattern, but it has many uses in real life.



1. Adapter – Implementation – Class Adapter:

It's time to see how we can implement the Adapter design pattern. Again, here I have some classes that I have already written to save us some time while implementing the Adapter design pattern. Now I will go over these classes one by one to see what they are. So first of all, we have an Employee class here and this is our Adaptee class. That means object of this class can provide the functionality that is needed by our client code. So what is this client code? So here we have a BusinessCardDesigner class, and this class is supposed to print business cards for its customers. It has a method called as designCard(), which expects an object which implements this Customer interface and it's going to call some methods on this object in order to print the business card. So what is Customer interface? So this is the Customer interface, which is the Target interface if you remember from our UML diagram and it has defined a few methods that are needed by our BusinessCardDesigner. Now, looking at our Employee class, we can see that the information required by Customer like name, designation and address is available in our Employee class, but that functionality is provided with a different method. So, inside Employee class, we have a getFullName() which can work where we expect the name, we have a getJobTitle() method that can work where the designation is required and we have a method called as getOfficeLocation() which can work where the address is required. So now we have a situation where we need an adapter in order to work with this code(i.e., Employee class code). So to start off, we are going to implement the 'Class Adapter', which is also called as a Two Way Adapter. So in order to implement the Class Adapter(i.e., EmployeeClassAdapter), we start by extending from our existing class, which is Employee. Then we need to also implement the interface that is expected by our client. So I'm going to say 'implements Customer'. It means that we now have to provide implementation for methods that are defined in our Customer interface. Now all we need to do is to translate these method calls{i.e., getName(), getDesignation(), getAddress()] to an appropriate method in our Adaptee(i.e., Employee). And we have already seen that getName() method, we have a method called as getFullName() in our Employee class that can work here. Now we are able to call this.getFullName() because our class(i.e., EmployeeClassAdapter) extends from Employee. So we have getFullName() method inherited in our adapter. On the similar lines, I'm going to quickly delegate the method calls to the appropriate method of Employee class. So this is our Class Adapter completely implemented. Now let's see how we can use this class(i.e., EmployeeClassAdapter) in our main() method. Now you will see that I have defined a method here called as populateEmployeeData(). This is a helper method that expects an Employee object and what it does is that it populates some sample data into that Employee type object so that we can use it to test out our code. So to begin with, we are first going to create an object of our Class Adapter(i.e., EmployeeClassAdapter). So I'm going to say EmployeeClassAdapter adapter = new EmployeeClassAdapter(). We don't need any constructor arguments here. Then we are going to make sure that we have an object of the client code, which is the BusinessCardDesigner. So if you remember, this is the class that we want to use, which expects a Customer class type object. So BusinessCardDesigner designer = new BusinessCardDesigner(), we're going to again create a simple object of this class and we are going to call a method on this 'designer' reference variable, which is the designCard(), and it expects a Customer object so we can pass our 'adapter' reference variable here. And then we are going to simply say print the 'card' reference variable so that we can make sure that this Adapter(i.e., EmployeeClassAdapter in our case) worked. Now, this is what Adapter design pattern does.

Object of our Adapter can be used where a Target interface type object is expected. But since we are using a 'Class Adapter', we can also use our Adapter object where our Adaptee is expected. So if you look at this populateEmployeeData() method, it expects an object of Employee class and our EmployeeClassAdapter since already extends Employee, we can also use our EmployeeClassAdapter object(i.e., 'adapter' reference variable) to call this populateEmployeeData() method. And this is why 'Class Adapter' is called as Two Way Adapter because if you look at line number 8 inside Main.java file, we are using our 'adapter' reference variable where an Adaptee is expected and we are using our 'adapter' reference variable on line number 10 where we have the designCard() method and here Customer interface is expected, there also we can use our 'adapter' reference variable. So that's why this is called as a Two Way Class Adapter.

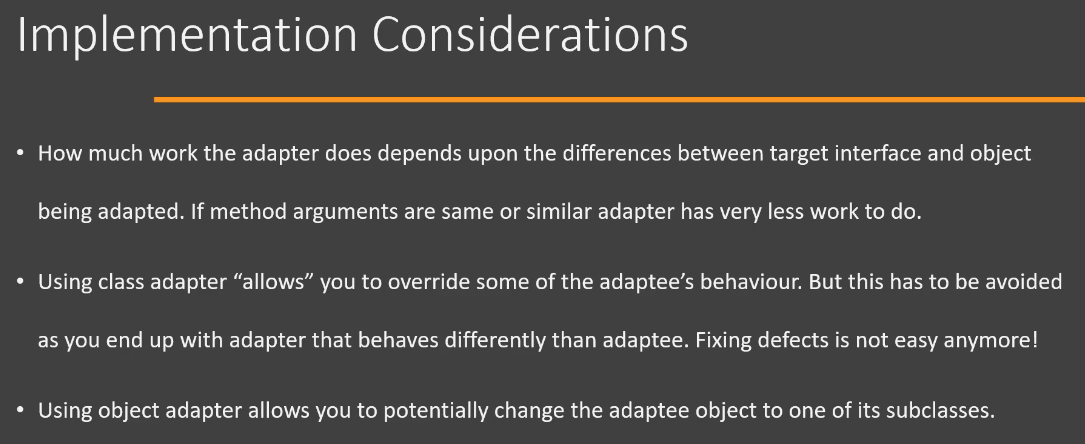
So this is how you can implement the Class Adapter or Two Way Adapter.

1. Adapter – Implementation – Object Adapter:

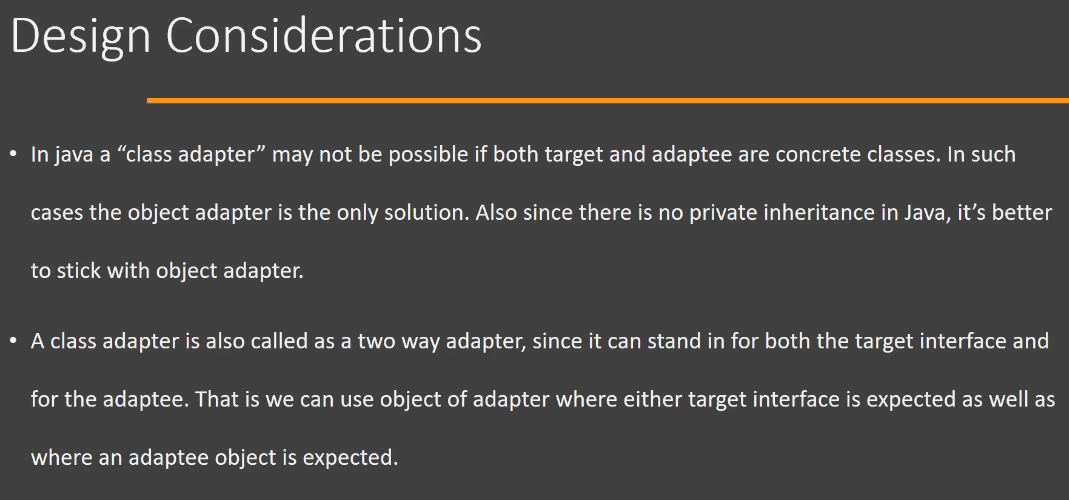
Now we're going to see how we can implement an Object Adapter in our Java code. So in the previous video, we have seen how we can implement a Two Way Class Adapter and now it's time to implement an Object Adapter. So here I have already created an empty class called as EmployeeObjectAdapter. So in order to implement an Object Adapter, we simply need to implement the Target(i.e., Customer) interface that is expected by our client code. So we simply implement Customer interface. We do not extend from our Employee class and then I will simply override the methods that are defined in our Customer interface. Now, an Object Adapter needs an object of our Adaptee(i.e., Employee). So I'm going to declare a private field where we are going to store the Adaptee(i.e., Employee) object/instance. So I'm going to name this object/instance as 'adaptee'. So this is the Adaptee(i.e., Employee) object that we have in our Object Adapter. And the method implementation itself is quite simple. Instead of using 'this' keyword, which we did in the Class Adapter, here we are going to call the same method but on the 'adaptee' object. So this is the only difference between Class Adapter and Object Adapter. Now the question remains how we are going to get hold of this Employee object inside the EmployeeObjectAdapter which we need. The best way to implement an Object Adapter is to write a constructor for your Object Adapter class(i.e., EmployeeObjectAdapter) and accept an Adaptee(i.e., Employee) object as your constructor argument. So now we have an Adaptee(i.e., Employee) object that we can use while implementing the methods that are defined in Target(i.e., Customer) interface. So you can see that this is quite simple. There is nothing to it. The only difference between Class Adapter and Object Adapter is we're going to use a composition in order to implement our functionality in Object Adapter. So let's jump over to main() method inside Main.java class and here we're going to use our Object Adapter. Now first of all, we're going to need an Employee object. So I'm going to create an object of Employee class, which is our Adaptee, because remember this object is needed by our Object Adapter. Now, before we can pass the Employee object to our adapter, I'm going to call the populate the employee data using populateEmployeeData() method. So this method is going to take an existing Employee object and populate some information that we can use. Next up, we need to create the object of our Object Adapter(i.e., EmployeeObjectAdapter). I'm going to name this reference variable as 'objectAdapter' and we're going to pass the Employee object that we just created. Now, the only thing that is remaining is we're going to again call the client code, which is going to expect a Customer object. We're going to call the designCard method of BusinessCardDesigner class and we're going to give it 'objectAdapter'. So once again, we simply created an Adaptee object. We get data populated in that object. We created our Object Adapter instance by providing it with our Adaptee object. And then we called our client code and passed on the Object Adapter instance. So this way we can implement and use our Object Adapter.

1. Adapter – Implementation & Design Considerations:

Now let's look at some of the implementation and design considerations when working with Adapter design pattern. The first point is that the amount of work that happens in the Adapter itself depends on how different our Target interface and our existing Adaptee class is. If they are quite similar and the method arguments are quite similar, then we can get away with simply method delegation. However, if the Target interface is accepting some arguments which are not compatible with our existing Adaptee class, we may need to do some translation and converting objects from one type to another in order to use our existing Adaptee class. So you should keep that in mind. Next, when you're using Class Adapter pattern, then you can technically override your Adaptee methods and change some of that behaviour. And it's quite tempting to do so if the situation demands. But you should avoid doing that because what you're going to end up with, is an Adapter that behaves differently from the Adaptee. So, the fixing of defects is not going to be a simple case. So stick with simple method delegation and any argument translation that is needed in the Adapter. Do not try to change the behaviour of Adaptee in class Adapter. Next using Object Adapter is beneficial, and the main reason for that is we can potentially change the Adaptee object that is used by our Object Adapter by one of its subclasses. So we may have a more refined subclass of Adaptee class in future that has some performance improvements or that may work more correctly. Then we can use object of that subclass instead of our original Adaptee, and that is possible only in Object Adapter.

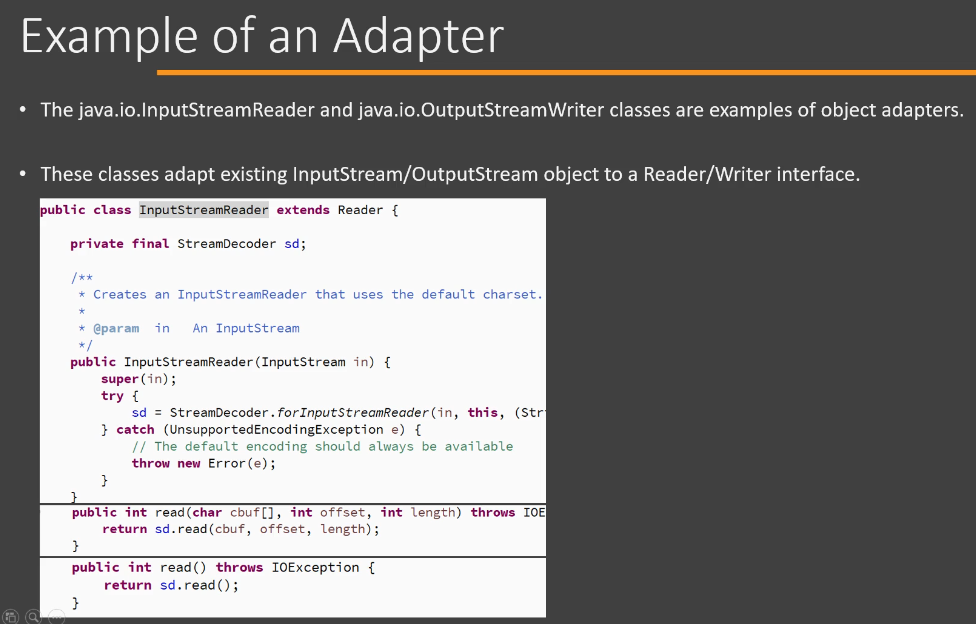


Now let's look at some of the design considerations. Now in Java, using Class Adapter may not even be possible in all cases because if you have a Target which is not an interface but a concrete class and you have your existing Adaptee class, then you cannot extend from both of these. So in those situations, the only way to implement this design pattern is using the Object Adapter. And since there is no concept of private inheritance, which is possible in C++, it's always a good idea to stick with Object Adapter. The reason for that, and which is the second point here, is Class Adapter is also called as Two Way Adapter. You may have heard me say this word and I wanted to hold on to that discussion until we had seen actual implementation. So what is a Two Way Adapter? It basically means that the Adapter class, since it extends from our Adaptee class and also implements the Target interface, object of that class(i.e., Adapter class) can work where the Target interface is expected, which we are already doing, but it can also work where our original Adaptee object is expected and that's what the Two Way Adapter term means. But this is not something that you want to do because since there is no private inheritance, your Adapter class ends up with methods that are not related because it has methods from the Target interface, it has methods from our Adaptee class, and you might end up polluting your codebase because you are exposing these unrelated methods and there is no way that you can stop someone from using these methods. So there are a lot of disadvantages in using Class Adapter. That's the verdict here.



1. Adapter – Real World Example:

Now let's look at an example of Adapter design pattern in the Java class library. And the best examples of that are the InputStreamReader and OutputStreamWriter classes, which are part of java.io package. And these classes are examples of Object Adapter, so these classes will adapt our existing InputStream or OutputStream objects(i.e., it can be thought of Adaptee class) to Reader or Writer abstract class(it can be thought of Target interface) which are also part of the Java IO package. So let's look at the code of the InputStreamReader. Now remember, this code is taken from the rt.jar file and you will see that we have a class here called as InputStreamReader. And in the constructor, we are giving an existing InputStream object to this adapter(i.e., InputStreamReader). Now we will see that this class is extending from the Reader class and this is how it is able to adapt to the Reader interface(specifically, it is an abstract class). So while providing the operation implementations for the operations[i.e., for example read()] that are defined in the Reader, you will see that our InputStreamReader is going to use StreamDecoder because sometimes Adapters have to do some additional work in order to adapt to the interface/abstract class. So using the StreamDecoder, it will delegate the operations to the underlying InputStream object. So InputStreamReader and OutputStreamWriter are examples of Object Adapter in Java class library.



A screenshot of a computer program

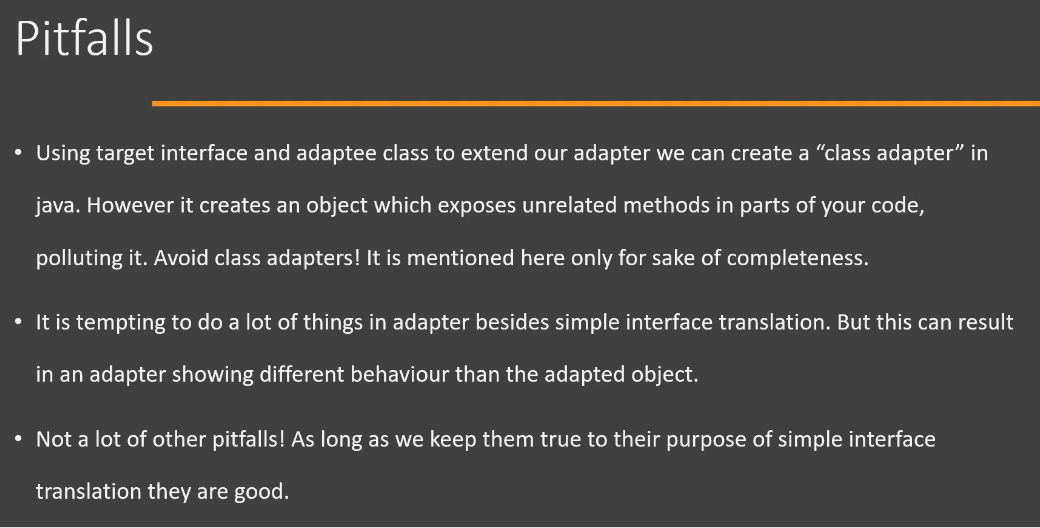
Description automatically generated

A screenshot of a phone

Description automatically generated

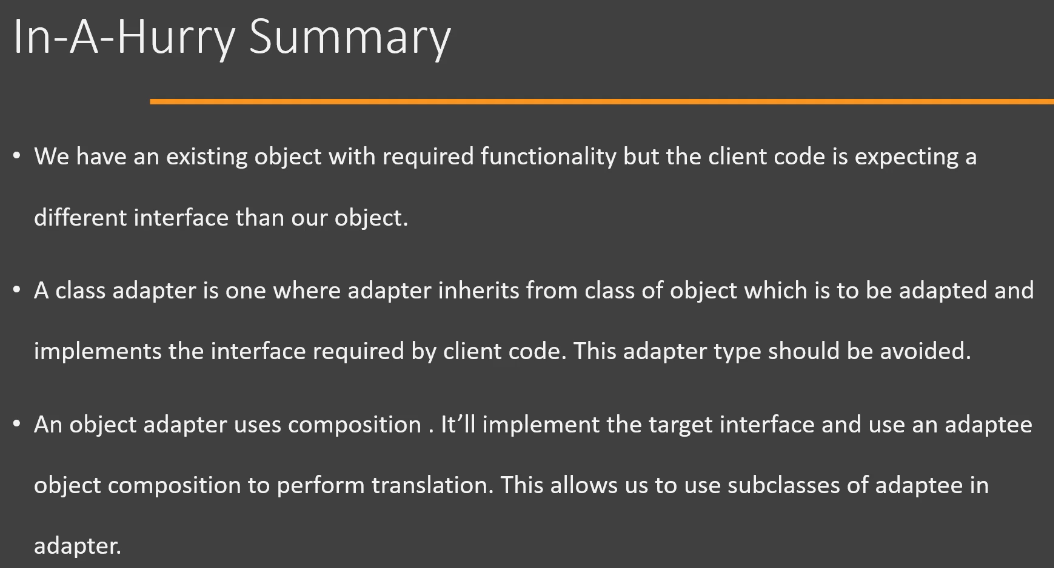
1. Adapter – Pitfalls:

Now let's look at the pitfalls of using Adapter design pattern. Now the first point here is related to Class Adapter, and I have been harping on this point since we started discussing this design pattern that Class Adapters are not a preferred way to implement Adapter design pattern. However, technically if you have a interface on one side, maybe a Target interface and your Adaptee is a non-final class, then you can technically implement a Class Adapter in Java. The reason I'm saying do not use Class Adapter is that you will Adapter class now has methods from Target interface since it has implemented that and it has methods from your Adaptee class. So you now have an object that has these sets of unrelated methods in it. And such an object, if used in your existing codebase, can end up polluting your code because people can start using methods from both Target interface and Adaptee class. One guy can use method from Target interface, another guy can use method from Adaptee class and it will become quite a nightmare. So that's one of the reason, main reason that you should avoid using Class Adapter. And a lot of times it may not even be possible to use Class Adapter because we only have a single inheritance in Java(Multiple Inheritance is not supported using classes). That means if there are classes on both sides(Target and Adaptee), then the Object Adapter is your only choice. Next, it is tempting to do a lot of things in your Adapter because now you have access to the method arguments that are going to the Adaptee object as well as you can access the output that is returned by the Adapter methods. So you may be tempted to do some validation, you may be tempted to do some modification or addition to the behaviour that is provided by our Adaptee class, but try to avoid any such temptation. Do not perform anything else in your Adapter apart from simple interface translation. And the reason for that is if your Adaptee class gets a defect fixed, then that defect fix may not be translated easily into your Adapter class, so your Adapter should remain faithful to whatever functionality/behaviour is offered by Adaptee. Do not even do validations here. Simply do interface translations. And there are no other pitfalls to this design pattern. It's perfectly good design pattern as long as you use it for its intended purpose and stick with only doing the interface translation, Adapter can be a really useful tool in your toolbox.



1. Adapter – Summary:

Now we're going to discuss summary of Adapter design pattern. So whenever we have an existing object(i.e., Adaptee) that provides the required functionality required by some part of code, but that particular code is expecting an object of different type of interface(i.e., Target type of object) than what our object(i.e., Adaptee) has implemented. In that case, we can use our Adapter design pattern. So there are two ways that you can implement this design pattern. The first one is called as a Class Adapter. In Class Adapter implementation, we inherit from our existing class(i.e., Adaptee) whose object we already have, and we implement the Target interface which is expected by the client code. This way we have a Class Adapter. Now this type of adapter should be avoided. Second way to implement our adapter is called as an Object Adapter and this adapter uses composition. So instead of extending from our existing class(i.e., Adaptee), this adapter will simply implement the interface(i.e., Target) which is expected by client code, and it will use the Adaptee object, the object that we already have as, in a composition in order to perform the operation which is requested by client. Now, one benefit of using Object Adapter is that we can substitute the object that is being used by adapter by one of its subclasses(i.e., object of one of its subclasses).



Now let's go over the UML diagram of the Adapter design pattern. So first of all, we're going to discuss the Class Adapter. So here we have an Adaptee class and this is the class whose object provides the functionality that is required by Client. Then we have a Target interface, which is what our Client expects. In order to implement Class Adapter, we extend from Adaptee and implement the Target interface. So we end up with a class which can be given to the Client and this class is going to simply delegate the operation that is requested by Client to method which it inherited from Adaptee. The next type of Adapter is the Object Adapter. So here again we have same arrangement. We have Adaptee class whose object provides the functionality that client needs, but it again expects this functionality to be provided in an object that implements this Target interface. So in Object Adapter, we simply implement the Target interface and we use the object of Adaptee in composition. And again, in the implementation of this Target interface, we simply delegate the operation to our Adaptee class object. so this is the Object Adapter.

A screenshot of a computer

Description automatically generated

A screenshot of a computer

Description automatically generated

Top of Form

Which one of the following scenarios is *most* suitable for using Adapter design pattern?

* **We have existing object that provides needed functionality but client code needs/expects different interface. Adapter works as interface translator. We adapt our object to new interface using Adapter.**