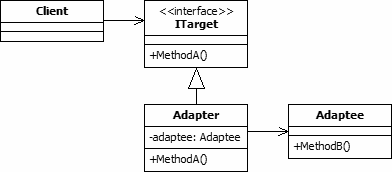
**Adapter Design (Structural) Pattern – 2022**

GOF : **Convert the interface of a class into another interface clients expect. Adapter lets classes work together that couldn't otherwise because of incompatible interfaces.**



There are three parts of the pattern:

1. **Client:** The class which has a reference of an [interface](https://www.educative.io/edpresso/what-is-a-java-interface) that it can recognize. Client will interact with Adapter using Target Interface
2. **Adaptee:** The unfamiliar class which the *client* needs to use. This is the existing interface which is wrapped by Adapter. Client wants to interact with Adaptee but cannot interact directly because Adaptee Interface is incompatible with Target Interface.
3. **Adapter:** A class introduced as an intermediary between the *client* and the *adaptee*. This class must implement the interface referenced by the *client* and also hold a reference to the *adaptee*. This is a wrapper over Adaptee class which implements the Target Interface. It receives calls from the client and translates that request to one/multiple adaptee calls using Adaptee interface.
4. **Target** Interface : This is the interface expected by the client.

**Important Points About Adapter Pattern**

* *Adapter class changes the interface of an existing object.*
* *Adapter class is a good example of object composition. Adapter class "has a" instance of the adaptee class.*
* *We can use an Adapter with any class Implementing Adaptee Interface.*
* *Adapter wraps an object to change it's interface whereas a decorator wraps an object to add extra functionalities.*

**When to Use Adapter Pattern**

* When an outside component provides captivating functionality that we'd like to reuse, but it's incompatible with our current application. A suitable Adapter can be developed to make them compatible with each other
* When our application is not compatible with the interface that our client is expecting
* When we want to reuse legacy code in our application without making any modification in the original code

**Example – 1**: An API provides data in json and another provides in XML, we can consume in Text format only

public class Client { 🡸 **Client**private TextFormatter format;  
  
 public Client(TextFormatter format) {  
 this.format = format;  
 }  
  
 public void process() {  
 String data = format.getText();  
 System.*out*.println("Client processing: "+data);  
 }  
}

public interface TextFormatter { 🡸 **Target**String getText();  
}

public class JsonApiData { 🡸 **Adaptee**public String getData() {  
 return "json";  
 }  
}

public class XmlAPIData { 🡸 **Adaptee**public String getData() {  
 return "XML";  
 }  
}

public class TextAdapter implements TextFormatter { 🡸 **Adapter**private JsonApiData jsonData;  
 public TextAdapter(JsonApiData jsonData) {  
 this.jsonData = jsonData;  
 }  
 public String getText() {  
 String tempData = jsonData.getData();  
 *// do some manipulation* return convert(tempData);  
 }  
 private String convert(String tempData) {  
 return "Text";  
 }  
}

**Test**

public static void main(String[] args) {  
 JsonApiData jsonData = new JsonApiData();  
 TextFormatter formatter = new TextAdapter(jsonData);  
 Client client = new Client(formatter);  
 client.process();  
}

Example-2: USBCable and CType cable

Understand the main method about the usage.

public static void main(String[] args) {  
 **MicroUSBClable usbCable = new USBCable();**  
 **CTypeCable cCableAdapter = new USB2CTypeAdapter(usbCable);** **IPhone iphone = new IPhone();  
 iphone.recharge(cCableAdapter);**}

public interface CTypeCable {  
 void recharge();  
}

public class IPhone {  
 public void recharge(CTypeCable cable) {  
 cable.recharge();  
 System.*out*.println("IPhone charging");  
 }  
}

🡺

public interface MicroUSBClable {  
 void recharge();  
}

public class AndroidPhone {  
 public void recharge(MicroUSBClable cable) {  
 cable.recharge();  
 }  
}

🡺

public class USBCableImpl implements MicroUSBClable {  
 @Override  
 public void recharge() {  
  
 }  
}

public class USB2CTypeAdapter implements CTypeCable { 🡸 Adapter  
 private MicroUSBClable microUsb;  
 public USB2CTypeAdapter(MicroUSBClable microUsb) {  
 this.microUsb = microUsb;  
 }  
  
 @Override  
 public void recharge() {  
 *// Convert to cType output* microUsb.recharge();  
 }  
}