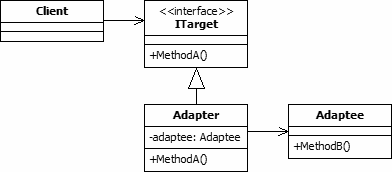
Adapter Design Pattern

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.**

# **Design Diagram**



**ITarget.java**

public interface ITarget {  
 void MethodA();  
}

Java Code as per the above diagram

**Client.java**

public class Client {  
 private ITarget \_target;  
  
 public Client(ITarget target) {  
 \_target = target;  
 }  
  
 public void MakeRequest() {  
 \_target.MethodA();  
 }  
}

**Adapter.java**

public class Adapter implements ITarget {  
 Adaptee \_adaptee = new Adaptee();  
  
 public void MethodA() {  
 \_adaptee.MethodB();  
 }  
}

**Adaptee.java**

public class Adaptee {  
 public void MethodB() {  
 System.*out*.println("MethodB called");  
 }  
}

**Test1.java**

public class Test1 {  
 public static void main(String[] args) {  
// ITarget target = new LegacyTarget();  
  
 ITarget target = new Adapter();  
  
 Client client = new Client(target); //Fixed  
  
 client.MakeRequest();//Fixed  
 }  
}

**LegacyTarget.java**

public class LegacyTarget implements ITarget {  
  
 @Override  
 public void MethodA() {  
 System.*out*.println("Legacy Method invocation ...");  
 }  
}

An example is given below. An existing system formats the string with new line character. A new system comes which can format the test with csv format.

Java code is given below.

**TextFormatter.java**

//Target  
public interface TextFormatter {  
   
 String formatText(String text);  
}

**Client1.java**

public class Client {  
  
 private TextFormatter formatter;  
  
 public Client(TextFormatter formatter) {  
 this.formatter = formatter;  
 }  
  
 public String format(String text) {  
 return formatter.formatText(text);  
 }  
}

**Test1.java**

public class Test1 {  
 public static void main(String[] args) {  
 String testString = " Formatting line 1. Formatting line 2. Formatting line 3.";  
 TextFormatter textFormatter = new NewLineTextFormatterImpl();  
 // This is as per legacy system  
 String resultString = textFormatter.formatText(testString);  
 System.*out*.println(resultString);  
 }  
}

**CSVFormatterImpl.java**

public class CSVFormatterImpl implements CSVFormatter {  
  
 @Override  
 public String formatCSV(String text) {  
 String formattedText=text.replace(".",",");  
 return formattedText;  
 }  
}

**CSVFormatter.java**

//Adaptee  
public interface CSVFormatter {  
   
 String formatCSV(String text);  
}

**CSVAdapterFormatter.java**

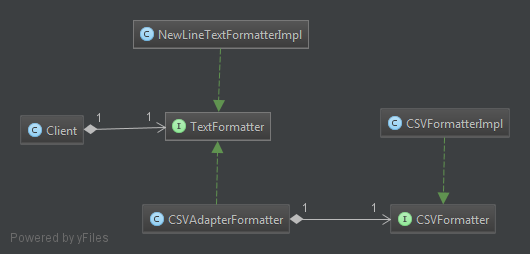
//Adapter  
public class CSVAdapterFormatter implements TextFormatter {  
   
 private CSVFormatter csvFormatter;  
   
 public CSVAdapterFormatter(CSVFormatter csvFormatter) {  
 this.csvFormatter = csvFormatter;  
 }  
  
 @Override  
 public String formatText(String text) {  
 String formattedText=csvFormatter.formatCSV(text);  
 return formattedText;  
 }  
}

The test program is given below.

**Test3.java**

public class Test3 {  
 public static void main(String[] args) {  
 CSVFormatter csvFormatter = new CSVFormatterImpl();  
 TextFormatter textFormatter = new CSVAdapterFormatter(csvFormatter);  
 String testString = " Formatting line 1. Formatting line 2. Formatting line 3.";  
  
 Client client = new Client(textFormatter);  
 String result = client.format(testString);  
 System.*out*.println("Final Result \n"+result);  
 }  
}

The class diagram is given below.



Another Example : Nokia Phone gets changed by Nokia cable and now we have apple cable, we need to use Adapter.

**NonUSBPlug.java**

//Adaptee Interface  
public interface NonUSBPlug {  
 public void connectDevice();  
}

**NokiaPlug.java**

public class NokiaPlug implements Plug {  
  
 @Override  
 public void connect() {  
 System.*out*.println("Nokia plug connected for charging ...");  
 }  
}

NonUSBPlug.java

//Adaptee Imlementation  
public class ApplePlug implements NonUSBPlug {  
  
 @Override  
 public void connectDevice() {  
 System.*out*.println("Device connected for charging ...");  
 }  
}

**MobileClient.java**

//It can be Cell Phone  
public class MobileClient {  
  
 private Plug plug;  
  
 public MobileClient(Plug plug) {  
 this.plug = plug;  
 }  
  
 public void charge() {  
 plug.connect();  
 }  
}

**Plug.java**

public interface Plug {  
 public void connect();  
}

**PlugAdapter.java**

//Adapter  
public class PlugAdapter implements Plug {  
  
 private NonUSBPlug plug;  
  
 public PlugAdapter( NonUSBPlug plug ) {  
 this.plug = plug;  
 }  
  
 @Override  
 public void connect() {  
 plug.connectDevice();  
 }  
}

The test program is given below.

public class Test1 {  
 public static void main(String[] args) {  
  
// NonUSBPlug nonUsbPlug = new ApplePlug();  
// Plug plug = new PlugAdapter(nonUsbPlug);  
  
 Plug plug = new NokiaPlug();  
 MobileClient client = new MobileClient(plug);  
 client.charge();  
 }  
}

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.
2. **Adaptee:** The unfamiliar class which the *client* needs to use.
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*.

## **Advantages of Adapter Pattern**

* *Two classes having incompatible interfaces can interact with each other using an Adapter class.*
* *It promotes reusability of existing system. A class can be accessed by multiple systems using different interfaces and Adapters.*

## **Components of Adapter Pattern**

* **Target Interface** : This is the interface expected by the client.
* **Adapter** : 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.
* **Adaptee Interface** : 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.
* **Client** : Client will interact with Adapter using Target Interface.

**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.*

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();  
}

Another Example

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 is getting charged ...");  
 }  
}

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();  
 }  
}