Adapter Design Pattern

##### **What is Adapter Design Pattern?**

In software development, it is common to encounter situations where existing classes or components have incompatible interfaces. The Adapter design pattern provides a solution by allowing objects with incompatible interfaces to work together seamlessly. By creating an adapter class that acts as an intermediary, the pattern enables collaboration between incompatible interfaces without modifying their underlying implementations.

That means the Adapter Design Pattern allows incompatible interfaces (objects) to work together. The Adapter Pattern acts as a bridge between two incompatible objects. Let’s say the first object is ABC and the second object is XYZ. And object ABC wants to consume some of the services provided by object XYZ. As these two objects are incompatible, they cannot communicate directly. In this case, Adapter will come into the picture and will act as a middleman or bridge between object ABC and object ZYZ. Now, object ABC will call the Adapter and Adapter will do the necessary transformations or conversions and then it will call object XYZ. So, Adapter which is a class is responsible for communication between two independent or incompatible interfaces.

The Adapter design pattern is a structural pattern that allows objects with incompatible interfaces to work together. It involves the following components:

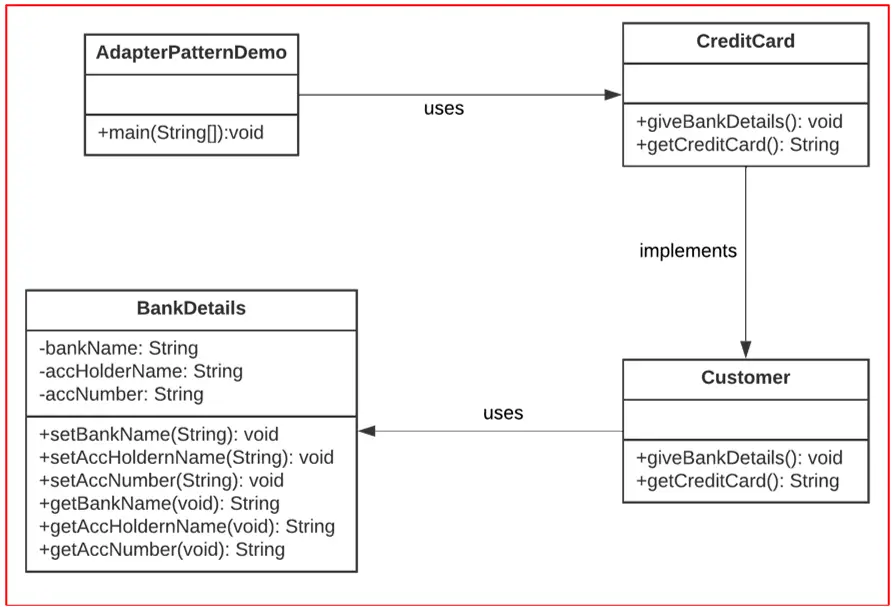
* **Target**: Defines the interface that the client code expects to interact with.
* **Adaptee**: Represents the existing class or component with an incompatible interface that needs to be adapted.
* **Adapter**: Acts as an intermediary between the client and the adaptee, implementing the target interface and delegating requests to the adaptee.

##### **Example to Understand Adapter Design Pattern in Java**

Let’s consider a real-world example where the Adapter pattern can be applied: connecting a new smartphone to an older audio system. In this scenario, the audio system has a traditional 3.5mm audio jack, while the smartphone only has a USB-C port. The Adapter pattern can be used to bridge the incompatibility and allow the smartphone to connect and play audio through the audio system.

The Adapter class will act as an intermediary between the smartphone and the audio system. It will implement the expected interface of the audio system and internally handle the conversion or translation of USB-C signals to the audio system’s requirements. The Adapter will have the necessary logic to convert the digital audio signal from the smartphone into an analog signal that the audio system can understand.

By using the Adapter pattern, the smartphone can seamlessly connect to the audio system without requiring any modifications to the audio system’s existing code or hardware. The Adapter acts as a translator, ensuring compatibility between the two systems and enabling them to work together. The UML Diagram of this example is given below using Adapter Design Pattern.

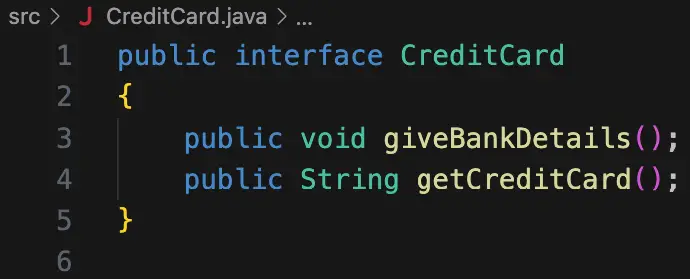


##### **Implementing Adapter Design Pattern in Java**

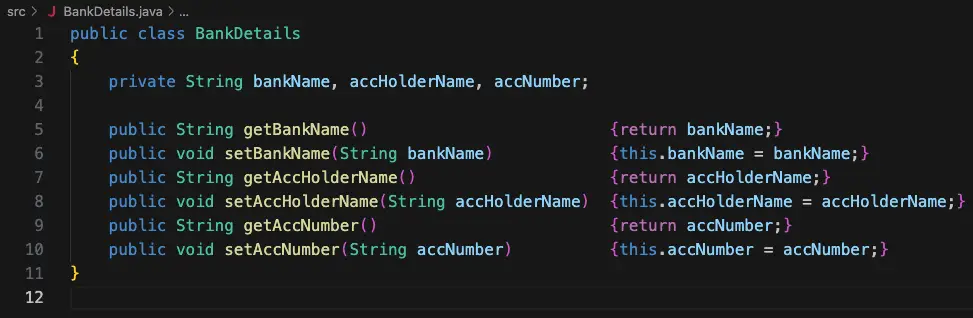
**Step 1:** Create a new directory to store all the class files of this project.

**Step 2:** Open VS Code and create a new project, called adapter.

**Step 3:** In the project, create a new file called CreditCard.java. Add the following code to the file:



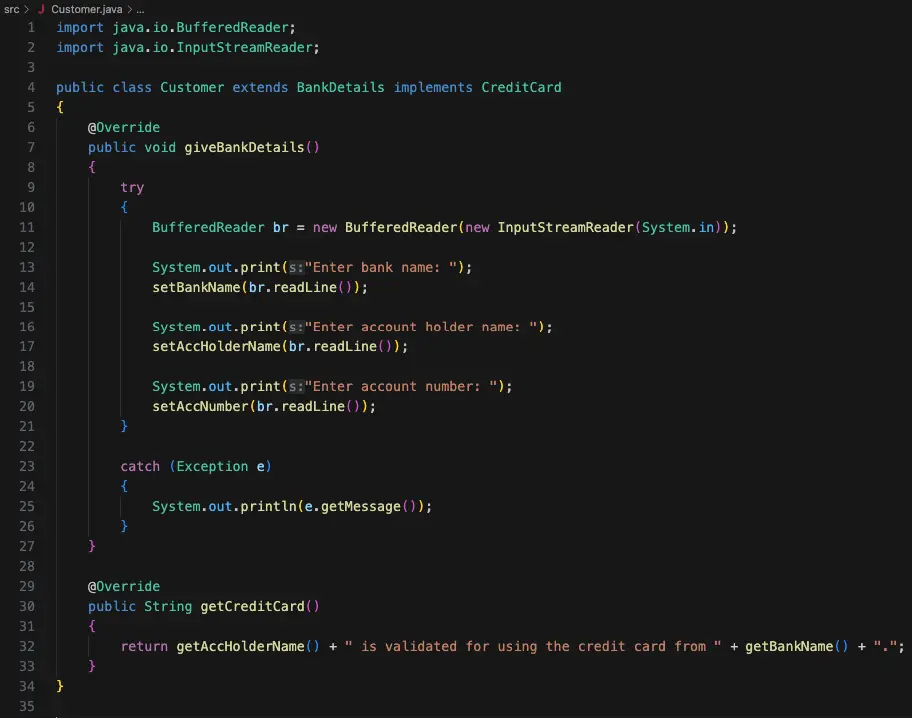
**Step 4:** In the project, create a new file called BankDetails.java. Add the following code to the file:



The setters and getters can easily be added by following these steps (this works for VS Code only; other IDEs usually have some similar option):

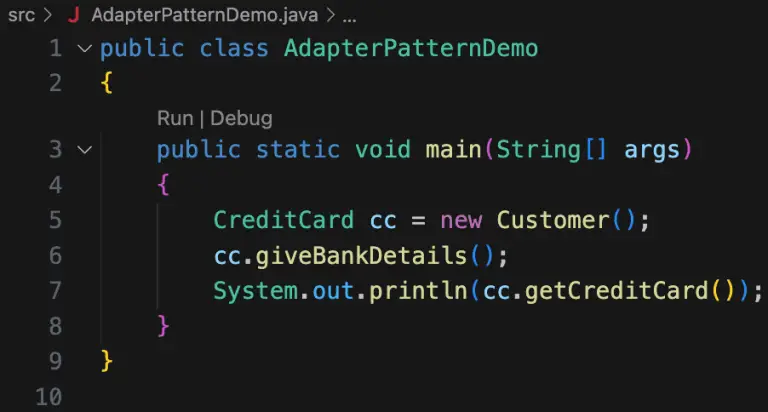
1. Add the fields to the class.
2. Right-click in the editor and select the “Source Action…” button.
3. In the menu that appears, select the “Generate Getters and Setters…” option.
4. In the drop-down menu that appears, select all three of the variables.

**Step 5:** In the project, create a new file called Customer.java. Add the following code to the file:



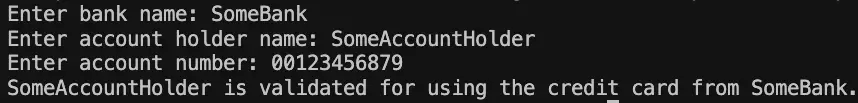
This class extends from BankDetails and implements the CreditCard interface. It implements the abstract methods from the CreditCard interface. It also uses the methods from the BankDetails class.

**Step 6:** In the project, create a new file called AdapterPatternDemo.java. This file will contain the main() function. Add the following code to the file:



This function creates a new object of type CreditCard. Then, it executes the two functions from the credit card interface (which were implemented in the Customer class).

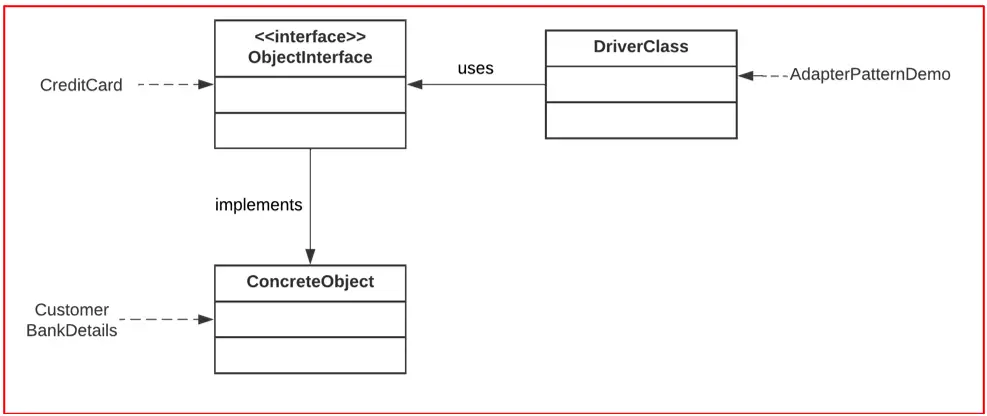
**Step 7:** Compile and execute the application. Ensure compilation is successful. Verify that the program works as expected.



Congratulations! You now know how to implement adapter patterns!

##### **UML Diagram of Adapter Design Pattern:**

Now, let us see the Adapter Design Pattern UML Diagram Components with our Example so that you can easily understand the UML Diagram.



The classes can be described as follows:

1. **ObjectInterface**: This is the interface that concrete classes can implement.
2. **ConcreteObject**: This object implements the aforementioned interface.
3. **DriverClass**: This class contains the main() function. It is responsible for handling the application simulation.

##### **The Complete Example Code of Adapter Design Pattern in Java**

###### **AdapterPatternDemo.java**

**public** **class** AdapterPatternDemo

**{**

**public** **static** **void** main**(String[]** args**)**

**{**

CreditCard cc = new Customer**()**;

cc.giveBankDetails**()**;

System.out.println**(**cc.getCreditCard**())**;

**}**

**}**

###### **BankDetails.java**

**public** **class** BankDetails

**{**

**private** **String** bankName, accHolderName, accNumber;

**public** **String** getBankName**()** **{return** bankName;**}**

**public** **void** setBankName**(String** bankName**)** **{**this.bankName = bankName;**}**

**public** **String** getAccHolderName**()** **{return** accHolderName;**}**

**public** **void** setAccHolderName**(String** accHolderName**)** **{**this.accHolderName = accHolderName;**}**

**public** **String** getAccNumber**()** **{return** accNumber;**}**

**public** **void** setAccNumber**(String** accNumber**)** **{**this.accNumber = accNumber;**}**

**}**

###### **CreditCard.java**

**public** **interface** CreditCard

**{**

**public** **void** giveBankDetails**()**;

**public** **String** getCreditCard**()**;

**}**

###### **Customer.java**

**import** *java.io.BufferedReader*;

**import** *java.io.InputStreamReader*;

**public** **class** Customer **extends** BankDetails **implements** CreditCard

**{**

@Override

**public** **void** giveBankDetails**()**

**{**

**try**

**{**

BufferedReader br = new BufferedReader**(**new InputStreamReader**(**System.in**))**;

System.out.print**(**"Enter bank name: "**)**;

setBankName**(**br.readLine**())**;

System.out.print**(**"Enter account holder name: "**)**;

setAccHolderName**(**br.readLine**())**;

System.out.print**(**"Enter account number: "**)**;

setAccNumber**(**br.readLine**())**;

**}**

**catch** **(**Exception e**)**

**{**

System.out.println**(**e.getMessage**())**;

**}**

**}**

@Override

**public** **String** getCreditCard**()**

**{**

**return** getAccHolderName**()** + " is validated for using the credit card from " + getBankName**()** + ".";

**}**

**}**

##### **Advantages of Adapter Design Pattern in Java**

Some of the advantages of using the adapter design pattern are:

1. **Code Reuse**: The Adapter pattern promotes code reuse by allowing the integration of existing classes or components into new systems without requiring significant modifications. Adapters encapsulate the necessary translations or mappings between interfaces, enabling the reuse of legacy code in modern systems.
2. **Interoperability**: The Adapter pattern facilitates interoperability by bridging incompatible interfaces. It allows components developed independently or by different teams to collaborate seamlessly, reducing the impact of interface incompatibilities and fostering system integration.
3. **System Extensibility**: The Adapter pattern enhances system extensibility by providing a flexible mechanism to incorporate new functionality. New adapters can be developed to integrate additional components or services without modifying existing code, minimizing the risk of introducing regressions or breaking existing functionality.
4. **Separation of Concerns**: Adapters promote the separation of concerns by isolating the logic required for interface translation. This separation allows for cleaner and more modular code, as each adapter focuses on a specific translation or adaptation task.
5. **Legacy System Integration**: The Adapter pattern is particularly useful when integrating legacy systems into modern architectures. Adapters can bridge the gap between legacy components and modern interfaces, allowing for a gradual migration and minimizing disruptions to the existing system.

##### **Disadvantages of Adapter Design Pattern in Java**

Some of the disadvantages of using the adapter design pattern are:

* **Increased Complexity**: The Adapter pattern introduces additional complexity to the codebase. Adapters may need to handle multiple translation scenarios, and the mapping between incompatible interfaces can become intricate. This complexity can impact code readability and maintainability.
* **Performance Overhead**: Adapters may introduce a performance overhead due to the translation and mapping operations required. These additional operations can impact system performance, especially in high-performance or time-critical scenarios. Careful consideration should be given to performance optimizations when using adapters.
* **Potential Overuse**: Overusing the Adapter pattern can lead to a bloated codebase with an excessive number of adapters. It is important to strike a balance and use the Adapter pattern judiciously when there is a genuine need to bridge incompatible interfaces.
* **Implicit Dependency**: Adapters may introduce implicit dependencies between the client code and the adaptee. This can hinder the ability to replace or switch adaptees without affecting the client code. Care should be taken to minimize tight coupling between adapters and clients.
* **Debugging and Troubleshooting**: When issues arise, debugging and troubleshooting can become more challenging due to the presence of adapters. The complexity introduced by adapters may require additional effort to trace and identify problems, especially when multiple adapters are involved in the system.

The Adapter design pattern offers significant advantages in facilitating code reuse, promoting interoperability, and enhancing system extensibility. It enables the collaboration between objects with incompatible interfaces, allowing for seamless integration and reducing the impact of interface incompatibilities. However, developers should consider potential disadvantages, including increased complexity, performance overhead, the risk of overuse, implicit dependencies, and potential challenges in debugging and troubleshooting. By understanding the specific requirements and trade-offs, developers can effectively leverage the Adapter pattern to achieve better code modularity, seamless integration, and increased flexibility in their software systems.