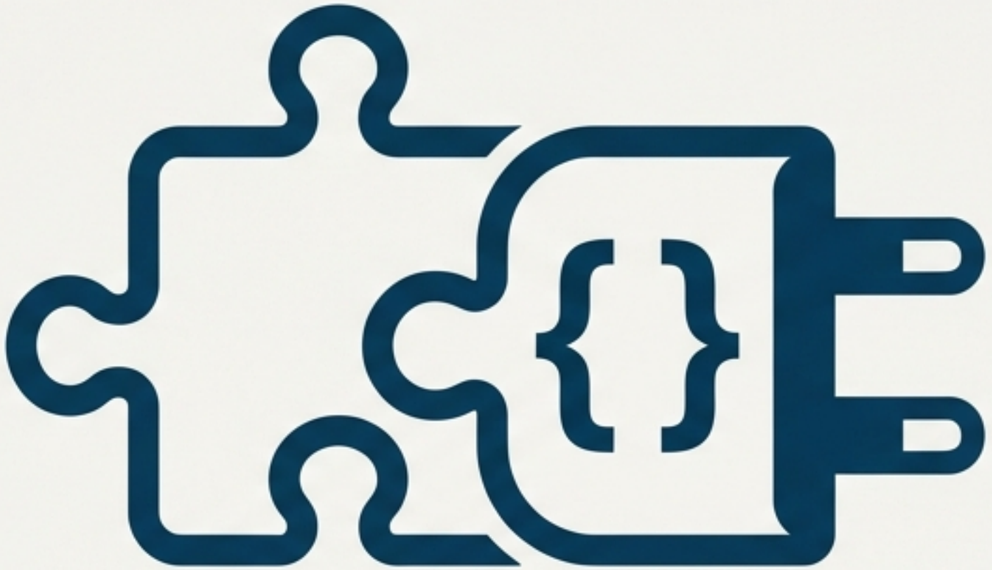


The Adapter Pattern

Making Incompatible Interfaces Work Together in Java

Incompatibility is a common problem.

Imagine you’re in a hotel room with US-type sockets, but your charger has an Indian plug. They are incompatible and cannot work together directly. The solution is an adapter that bridges this gap.

Our software faces the exact same challenge.

Our Existing Application (`Client`)

Your application's code expects to work with a specific interface. However, a useful third-party library you need to integrate has a completely different interface. They are like two puzzle pieces that don't fit.

The Adapter Pattern creates a bridge between worlds.

The Adapter converts the interface of a class into another interface that clients expect. It lets classes work together that couldn’t otherwise because of incompatible interfaces.

A Practical Case Study: The Report Generator

- **Our Client's Goal:** Our application needs to fetch report data and process it in a clean **JSON** format.
- **The Available Tool:** We have a third-party library, XmlDataProvider, that provides the necessary data, but only in **XML** format.

The Target: What Our Client Code Expects

The `Client` is coded to work with this specific `IReports` interface. It is the "socket" that our solution must plug into. It defines the contract that must be fulfilled.

```
// This is the Target Interface our Client expects.
public interface IReports {
    String getJsonData(String rawData);
}
```

The Adaptee: The Incompatible but Useful Library

This `XmlDataProvider` class contains the data logic we need, but its method signature does not match our `IReports` interface. This is our 'foreign plug.'

```
// This is the Adaptee — the third-party library.
public class XmlDataProvider {
    public String getXmlData(String data) {
        // ... internal logic to convert raw data to XML ...
        return "<report><name>Alice</name><id>42</id></report>";
    }
}
```

Building the Bridge: The `XmlDataProviderAdapter`

The Adapter's structure is key. It simultaneously satisfies the client's contract and wraps the incompatible library.

1. **`is-a` Target***: It `implements` the `IReports` interface, so the Client can use it.
2. **`has-a` Adaptee***: It holds a reference to the `XmlDataProvider` to delegate the actual work.

`is-a` relationship via implementation

`has-a` relationship via composition

```
// The Adapter class
public class XmlDataProviderAdapter implements IReports {

    private XmlDataProvider provider;

    public XmlDataProviderAdapter(XmlDataProvider provider) {
        this.provider = provider;
    }

    @Override
    public String getJsonData(String rawData) {
        // ... the translation logic will go here ...
        return "";
    }
}
```

Inside the Adapter: The Translation Logic

Within the `getJsonData` method, the Adapter orchestrates a two-step translation:

1. It delegates the initial data retrieval call to the Adaptee's method.
2. It converts the returned XML data into the JSON format that the Client expects.

```
@Override
public String getJsonData(String rawData) {
    // 1. Delegate the call to the adaptee to get XML data.
    String xmlData = provider.getXmlData(rawData);

    // 2. Convert the XML result to the target JSON format.
    // (In a real app, this would use a robust XML/JSON library).
    String jsonData = convertXmlToJson(xmlData);

    // 3. Return the data in the format the client expects.
    return jsonData;
}
```

The Client's View: Decoupled and Unaware

The `Client` code only interacts with the `IReports` interface. It has no knowledge of the `XmlDataProvider` or the XML-to-JSON conversion. This means we can change or replace the data provider without ever touching the client code.

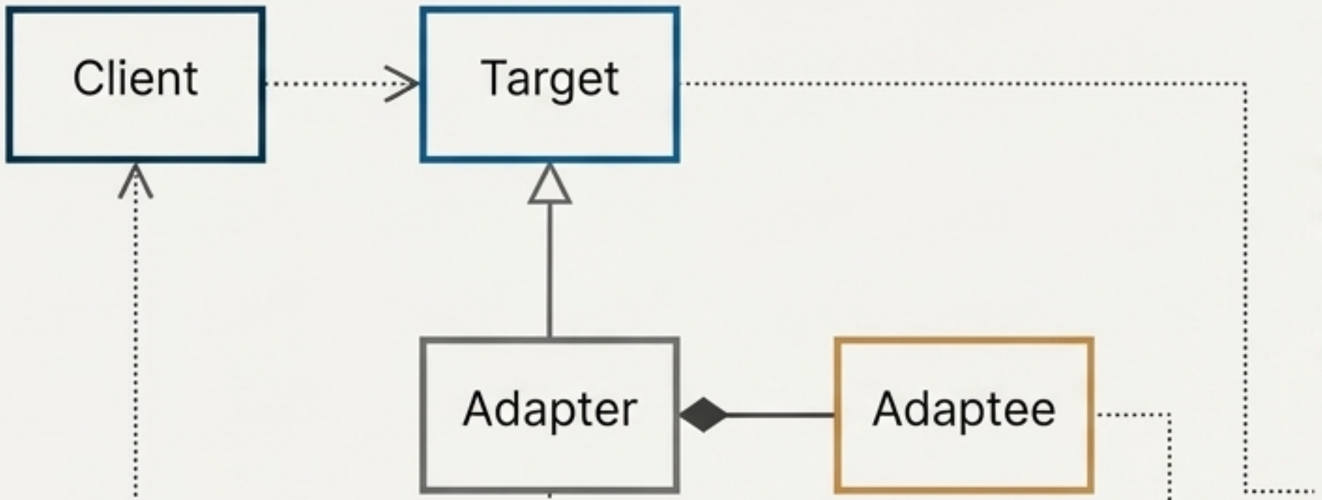
```
public static void main(String[] args) {
    // 1. Create the incompatible Adaptee object.
    XmlDataProvider adaptee = new XmlDataProvider();

    // 2. Create the Adapter and pass it the Adaptee.
    IReports adapter = new XmlDataProviderAdapter(adaptee);

    // 3. The Client uses the Adapter via the Target interface.
    Client client = new Client();
    String jsonResult = client.getReport(adapter, "Alice:42");

    System.out.println(jsonResult);
    // Output: {"name":"Alice", "id":42}
}
```

The Pattern's Formal Blueprint



Our implementation maps directly to the standard UML diagram for the Adapter pattern. This structure is a recognized solution to the problem of incompatible interfaces.

- **Target**: `IReports`
- **Adapter**: `XmlDataProviderAdapter`
- **Adaptee**: `XmlDataProvider`
- **Client**: `Client` (our `main` method)

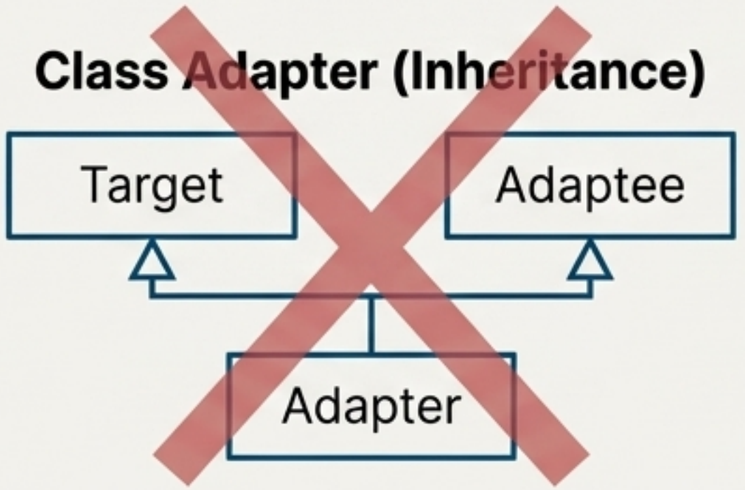
A Key Distinction: Object vs. Class Adapters

- There are two main flavors of the Adapter pattern:
  - **Object Adapter** (Used in our example**):** Uses **Composition** (`has-a`). The Adapter holds an instance of the Adaptee. This is flexible and the standard approach in Java.
  - **Class Adapter: Uses Multiple Inheritance**. The Adapter inherits from both the Target (interface) and the Adaptee (class). This is less flexible and **not directly possible in Java** with classes.

In Java and modern software design, we always **prefer composition over inheritance**.

Object Adapter (Composition)

Class Adapter (Inheritance)



When Should You Use the Adapter Pattern?

Integrating Third-Party Libraries

The most common use case. An external library or vendor API has a different interface than what your system expects.

Working with Legacy Code

When a modern application needs to communicate with an older system that cannot be modified. The adapter can translate modern calls into the legacy system's format.

Creating a Unified Interface

When you have multiple subsystems or classes with slightly different functionalities that you want to use through a single, standardized interface.

The Adapter Pattern: Core Principles

- **Purpose**: To make incompatible interfaces **compatible**, allowing components to work together.
- **Structure**: The `Adapter` class `implements` a `Target` interface and is **composed of** an `Adaptee` object.
- **Core Benefit**: It **decouples** the `Client` from the `Adaptee`'s specific implementation. This makes your system more flexible and easier to maintain.
- **The Java Way**: Use the Object Adapter pattern, favoring **composition over inheritance**.

Thank You

Questions?