# Design Pattern: Adapter

The adapter pattern is a structural design pattern, that allows objects with incompatible interfaces to work together. It acts as a bridge between two incompatible interfaces, converting the interface of one type into another interface that the client expects. This pattern is useful when you have existing code or classes that cannot be easily modified to work with each other due to incompatible interfaces.

Adapter solves the problem of having incompatible interfaces. It allows the code to work with classes that don't have compatible interfaces. It also allows to reuse existing classes that don't implement the required interface.

The Adapter pattern consists of the following key components:

Target: This is the desired interface that the client code expects to work with. It defines the methods or operations that the client can use.

Adapter: The adapter class implements the Target interface and wraps an instance of the Adaptee. It acts as a mediator between the client and the Adaptee, converting the requests from the client to the format or interface expected by the Adaptee.

Adaptee: The Adaptee is the existing class or interface that needs to be adapted to work with the client code. It has a different interface or set of methods than what the client expects.

To use the Adapter pattern, the client interacts with the Target interface, which internally delegates the requests to the Adapter. The Adapter, translates these requests into a format that the Adaptee can understand and performs the necessary operations. The client remains unaware of the Adaptee's existence.

The Adapter pattern helps in achieving reusability by allowing classes with different interfaces to work together. It is often used when integrating legacy systems or when working with third-party libraries that have incompatible interfaces.

---

With adapters you don't want to change the behavior, you just want to have it to pass a request.

## Example

The example below shows how to use the adapter pattern to convert the interface of a class into another interface that another part of the code expect.

```
public class AdapterPatternExample {

    public static void main(String[] args) {
        // Create a new instance of the adapter
        ITarget adapter = new Adapter();

        // Call the method that is expected by the client
        adapter.request();
    }
}
```

```
// The target defines the domain-specific interface that the client uses
interface ITarget {
    void request();
}
```

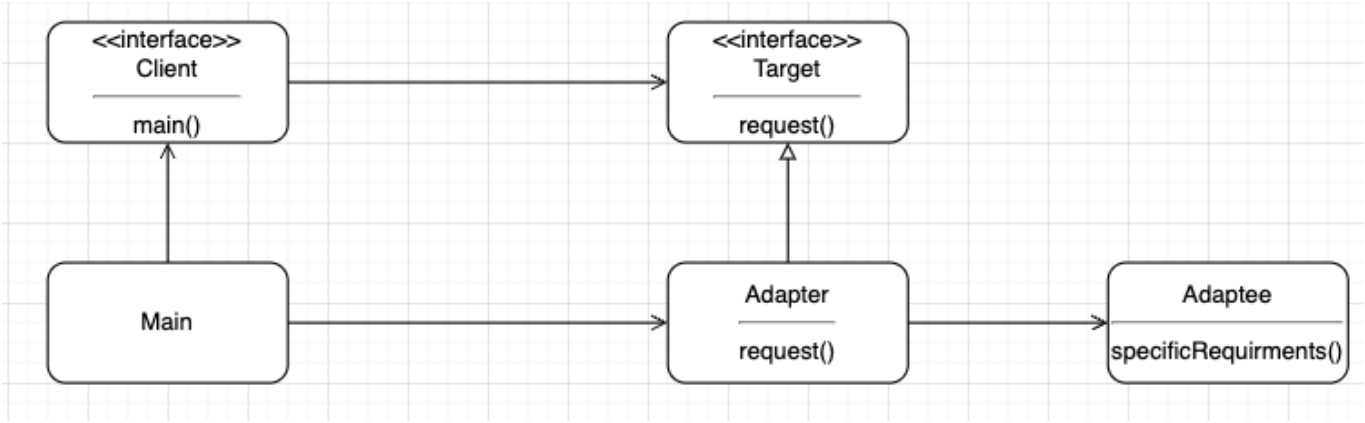
```
// The adaptee contains some useful behavior, but its interface is
incompatible with the existing client code

class Adaptee {
    public void specificRequest() {
        System.out.println("Called specificRequest()");
    }
}
```

```
// The adapter makes the adaptee's interface compatible with the target's
interface
`
class Adapter implements ITarget {`
    private Adaptee adaptee = new Adaptee();

    public void request() {
        adaptee.specificRequest();
    }
}
```

## UML Diagram




---

## When to use the Adapter pattern

Say there is an application that manages a collection of shapes, represented by a Shape interface with methods like draw() and resize(). The application works well with the existing shape classes such as Circle,

Rectangle, and Triangle.

Then you want to increase the shape functionality by adding a new class called Square. However, the Square class has a different interface than the existing shape classes. It has methods like render() and scale() instead of draw() and resize(). To use the functionality of the ExternalShape within your application, you can employ the Adapter pattern. Here's how the implementation might look:

Define the Shape interface with methods like draw() and resize(). This is your existing interface that the application is built around.

Create an ExternalShapeAdapter class that implements the Shape interface. This class will act as the adapter between the existing application code and the ExternalShape interface.

In the ExternalShapeAdapter, maintain a reference to an instance of the ExternalShape. This allows the adapter to delegate the method calls to the appropriate methods of the ExternalShape.

Implement the methods of the Shape interface in the ExternalShapeAdapter by internally invoking the corresponding methods of the ExternalShape. For example, the draw() method of the ExternalShapeAdapter can call the render() method of the ExternalShape.

With this setup, your existing application code can work seamlessly with the ExternalShape library by treating the ExternalShapeAdapter objects as regular Shape objects. The adapter converts the calls from the existing code to the appropriate calls expected by the ExternalShape.

---

## 4 pillars of OOP

Encapsulation: The Adapter pattern encapsulates the interaction between the client and the Adaptee within the Adapter class. The client interacts with the Adapter using the Target interface, which hides the complexities of the Adaptee's interface and implementation.

Inheritance/Polymorphism: The Adapter class inherits from the Target interface and implements its methods. This allows the Adapter to be treated as an instance of the Target interface, enabling polymorphic behavior. The Adapter also internally uses an instance of the Adaptee class, allowing it to utilize inheritance and polymorphism.

Abstraction: The Adapter pattern provides an abstraction layer between the client and the Adaptee. The client only needs to work with the Target interface, abstracting away the details of the Adaptee's implementation. This abstraction allows for code modularity and reduces dependencies between the client and the Adaptee.