Adapter Pattern - Complete Guide

🎯 Pattern Overview

The Adapter Pattern is a structural design pattern that makes **incompatible interfaces compatible**. This pattern works like a wrapper that helps two different classes work together seamlessly.

In simple terms: **A bridge between old and new code!**

🧩 Core Components

1. Target Interface

- This is the interface that the **client expects**
- The standard interface of the new system
- What the client wants to use directly

2. Adaptee (Legacy Code)

- This is the **existing class** that's already implemented
- Its interface is different from the Target
- Cannot be modified (third-party library or legacy code)

3. Adapter

- **Works as a bridge**
- Implements the Target interface
- Uses the Adaptee internally
- Acts as a translator between Client and Adaptee

4. Client

- Uses the Target interface
- Accesses the Adaptee through the Adapter
- Doesn't know about the Adaptee directly

🏠 Real-Life Example: Phone Charger System

```
Phone (Client)
   ↓
USB-C Port (Target Interface)
   ↓
USB-C to Lightning Adapter (Adapter)
   ↓
Lightning Cable (Adaptee/Legacy)
   ↓
iPhone (Works!)
```

Another Example: Media Player System

```
Media Player App (Client)
   ↓
MediaPlayer Interface (Target)
   ↓
AudioAdapter (Adapter)
   ↓
AdvancedMediaPlayer (Adaptee)
   ↓
MP4/VLC Player (Legacy Systems)
```

🔄 Working Flow

1. **Client Request**: Client makes a request through the Target interface
2. **Adapter Translation**: Adapter receives the request
3. **Method Mapping**: Adapter calls the corresponding method on the Adaptee
4. **Response Translation**: Adaptee's response is converted back to Target format
5. **Client Response**: Client receives the response in the expected format

✅ Advantages

1. Legacy Code Reuse

- Use old code without modification
- Protects existing investments

2. Third-Party Integration

- Easily integrate external libraries
- Hide API differences

3. Separation of Concerns

- Business logic and interface conversion are separate
- Maintains clean architecture

4. **Open/Closed Principle**

- Can add new adapters
- No need to modify existing code

5. **Multiple Adaptees Support**

- One adapter can support multiple legacy systems

❌ Disadvantages

1. **Code Complexity**

- Adds an extra abstraction layer
- Can make debugging more difficult

2. **Performance Overhead**

- Extra layer for method calls
- Increased memory footprint

3. **Maintenance**

- Need to maintain adapter code
- Must track changes on both sides

🛠️ Implementation Example

Object Adapter Pattern

```java
```

```
// Target Interface - What the client wants
interface MediaPlayer {
    void play(String audioType, String fileName);
}

// Adaptee - Legacy/Third-party classes
interface AdvancedMediaPlayer {
    void playVlc(String fileName);
    void playMp4(String fileName);
}

class VlcPlayer implements AdvancedMediaPlayer {
    @Override
    public void playVlc(String fileName) {
        System.out.println("Playing vlc file: " + fileName);
    }

    @Override
    public void playMp4(String fileName) {
        // Empty - VLC player doesn't support MP4
    }
}

class Mp4Player implements AdvancedMediaPlayer {
    @Override
    public void playVlc(String fileName) {
        // Empty - MP4 player doesn't support VLC
    }

    @Override
    public void playMp4(String fileName) {
        System.out.println("Playing mp4 file: " + fileName);
    }
}

// Adapter - Bridge between Target and Adaptee
class MediaAdapter implements MediaPlayer {
    private AdvancedMediaPlayer advancedMusicPlayer;

    public MediaAdapter(String audioType) {
        if (audioType.equalsIgnoreCase("vlc")) {
            advancedMusicPlayer = new VlcPlayer();
        } else if (audioType.equalsIgnoreCase("mp4")) {
            advancedMusicPlayer = new Mp4Player();
        }
    }
```

```
    @Override
    public void play(String audioType, String fileName) {
        if (audioType.equalsIgnoreCase("vlc")) {
            advancedMusicPlayer.playVlc(fileName);
        } else if (audioType.equalsIgnoreCase("mp4")) {
            advancedMusicPlayer.playMp4(fileName);
        }
    }
}

// Client
class AudioPlayer implements MediaPlayer {
    private MediaAdapter mediaAdapter;

    @Override
    public void play(String audioType, String fileName) {
        // Built-in support for MP3
        if (audioType.equalsIgnoreCase("mp3")) {
            System.out.println("Playing mp3 file: " + fileName);
        }
        // Use adapter for other formats
        else if (audioType.equalsIgnoreCase("vlc") ||
                 audioType.equalsIgnoreCase("mp4")) {
            mediaAdapter = new MediaAdapter(audioType);
            mediaAdapter.play(audioType, fileName);
        } else {
            System.out.println("Invalid media. " + audioType +
                               " format not supported");
        }
    }
}

// Usage
public class AdapterPatternDemo {
    public static void main(String[] args) {
        AudioPlayer audioPlayer = new AudioPlayer();

        audioPlayer.play("mp3", "beyond_the_horizon.mp3");
        audioPlayer.play("mp4", "alone.mp4");
        audioPlayer.play("vlc", "far_far_away.vlc");
        audioPlayer.play("avi", "mind_me.avi"); // Not supported
    }
}
```

Class Adapter Pattern (Using Inheritance)

java

```java
// Target Interface
interface Rectangle {
    void draw(int x, int y, int width, int height);
}

// Adaptee (Legacy class)
class LegacyRectangle {
    public void draw(int x1, int y1, int x2, int y2) {
        System.out.println("Drawing rectangle from (" + x1 + "," + y1 +
                           ") to (" + x2 + "," + y2 + ")");
    }
}

// Class Adapter
class RectangleAdapter extends LegacyRectangle implements Rectangle {
    @Override
    public void draw(int x, int y, int width, int height) {
        // Convert coordinate system
        int x2 = x + width;
        int y2 = y + height;
        super.draw(x, y, x2, y2);
    }
}

// Client
public class ClassAdapterDemo {
    public static void main(String[] args) {
        Rectangle rectangle = new RectangleAdapter();
        rectangle.draw(10, 20, 100, 50);
    }
}
```

🔧 Advanced Implementation Patterns

1. Generic Adapter

java

```java
public class GenericAdapter<T, U> {
    private final U adaptee;
    private final Function<T, Object[]> requestMapper;
    private final Function<Object, T> responseMapper;

    public GenericAdapter(U adaptee,
                          Function<T, Object[]> requestMapper,
                          Function<Object, T> responseMapper) {
        this.adaptee = adaptee;
        this.requestMapper = requestMapper;
        this.responseMapper = responseMapper;
    }

    public T adapt(T input, String methodName) throws Exception {
        Object[] params = requestMapper.apply(input);
        Method method = adaptee.getClass().getMethod(methodName,
                                        getParameterTypes(params));
        Object result = method.invoke(adaptee, params);
        return responseMapper.apply(result);
    }

    private Class<?>[] getParameterTypes(Object[] params) {
        return Arrays.stream(params)
                .map(Object::getClass)
                .toArray(Class<?>[]::new);
    }
}
```

2. Configuration-Based Adapter

```java
```

```
@Configuration
public class AdapterConfig {

    @Bean
    public MediaPlayer createMediaPlayer(@Value("${media.type}") String type) {
        switch (type.toLowerCase()) {
            case "advanced":
                return new MediaAdapter("vlc");
            case "basic":
                return new BasicMediaPlayer();
            default:
                return new DefaultMediaPlayer();
        }
    }
}
```

3. Caching Adapter

java

```
class CachingAdapter implements MediaPlayer {
    private final Map<String, Object> cache = new ConcurrentHashMap<>();
    private final MediaAdapter adapter;

    public CachingAdapter(MediaAdapter adapter) {
        this.adapter = adapter;
    }

    @Override
    public void play(String audioType, String fileName) {
        String cacheKey = audioType + ":" + fileName;

        if (!cache.containsKey(cacheKey)) {
            adapter.play(audioType, fileName);
            cache.put(cacheKey, "played");
        } else {
            System.out.println("Playing from cache: " + fileName);
        }
    }
}
```

🎯 When to Use Adapter Pattern

✅ Use When:

- **Legacy system integration** is needed

- **Third-party libraries** have incompatible interfaces
- **Different data formats** need to be supported
- **API versioning** issues exist
- **Existing code** needs to be reused without modification

❌ Avoid When:

- **Simple interface** differences (just method renaming)
- **Performance** critical applications where overhead isn't acceptable
- **Direct modification** is possible and safe
- **Over-engineering** is occurring

💥 Real-World Examples

1. Database Adapters

java

```java
// JDBC drivers for different databases
interface DatabaseConnection {
    void connect();
    ResultSet executeQuery(String query);
}

class MySQLAdapter implements DatabaseConnection {
    private MySQLDriver mysqlDriver;

    @Override
    public void connect() {
        mysqlDriver.establishConnection();
    }

    @Override
    public ResultSet executeQuery(String query) {
        return mysqlDriver.runSQL(query);
    }
}
```

2. Payment Gateway Integration

java

```java
interface PaymentProcessor {
    PaymentResult processPayment(double amount, String currency);
}

class PayPalAdapter implements PaymentProcessor {
    private PayPalAPI paypalAPI;

    @Override
    public PaymentResult processPayment(double amount, String currency) {
        PayPalRequest request = new PayPalRequest(amount, currency);
        PayPalResponse response = paypalAPI.makePayment(request);
        return convertToPaymentResult(response);
    }
}

class StripeAdapter implements PaymentProcessor {
    private StripeClient stripeClient;

    @Override
    public PaymentResult processPayment(double amount, String currency) {
        StripeCharge charge = stripeClient.createCharge(
            (int)(amount * 100), currency); // Stripe uses cents
        return convertToPaymentResult(charge);
    }
}
```

3. Social Media Integration

java

```
interface SocialMediaPoster {
    void post(String message, String imageUrl);
}

class TwitterAdapter implements SocialMediaPoster {
    private TwitterAPI twitterAPI;

    @Override
    public void post(String message, String imageUrl) {
        Tweet tweet = new Tweet();
        tweet.setText(message.substring(0, Math.min(message.length(), 280)));
        tweet.setMediaUrl(imageUrl);
        twitterAPI.postTweet(tweet);
    }
}

class InstagramAdapter implements SocialMediaPoster {
    private InstagramAPI instagramAPI;

    @Override
    public void post(String message, String imageUrl) {
        InstagramPost post = new InstagramPost();
        post.setCaption(message);
        post.setImageUrl(imageUrl);
        instagramAPI.uploadPost(post);
    }
}
```

4. File Format Conversion

java

```
interface DocumentReader {
    String readDocument(String filePath);
}

class PDFAdapter implements DocumentReader {
    private PDFBoxLibrary pdfBox;

    @Override
    public String readDocument(String filePath) {
        PDFDocument doc = pdfBox.loadDocument(filePath);
        return pdfBox.extractText(doc);
    }
}

class WordAdapter implements DocumentReader {
    private ApachePOI apachePOI;

    @Override
    public String readDocument(String filePath) {
        XWPFDocument doc = apachePOI.openDocument(filePath);
        return apachePOI.extractText(doc);
    }
}
```

5. REST to SOAP Adapter

java

```
interface RESTService {
    String getData(String id);
    void saveData(String id, Object data);
}

class SOAPAdapter implements RESTService {
    private SOAPClient soapClient;

    @Override
    public String getData(String id) {
        SOAPRequest request = createSOAPRequest("GET_DATA", id);
        SOAPResponse response = soapClient.call(request);
        return parseResponse(response);
    }

    @Override
    public void saveData(String id, Object data) {
        SOAPRequest request = createSOAPRequest("SAVE_DATA", id, data);
        soapClient.call(request);
    }
}
```

📊 Performance Considerations

1. Lazy Initialization

java

```
class LazyAdapter implements MediaPlayer {
    private MediaAdapter adapter;
    private final String audioType;

    public LazyAdapter(String audioType) {
        this.audioType = audioType;
    }

    @Override
    public void play(String audioType, String fileName) {
        if (adapter == null) {
            adapter = new MediaAdapter(audioType);
        }
        adapter.play(audioType, fileName);
    }
}
```

2. Connection Pooling

java

```
class PooledDatabaseAdapter implements DatabaseConnection {
    private final ConnectionPool pool;

    @Override
    public ResultSet executeQuery(String query) {
        Connection conn = pool.getConnection();
        try {
            return conn.executeQuery(query);
        } finally {
            pool.releaseConnection(conn);
        }
    }
}
```

3. Batch Processing

java

```
class BatchAdapter implements DataProcessor {
    private final LegacyProcessor processor;
    private final List<DataItem> batch = new ArrayList<>();
    private final int batchSize;

    @Override
    public void process(DataItem item) {
        batch.add(item);
        if (batch.size() >= batchSize) {
            processBatch();
        }
    }

    private void processBatch() {
        processor.processBatch(batch);
        batch.clear();
    }
}
```

🎓 Best Practices

1. Error Handling

java

```
class RobustAdapter implements MediaPlayer {
    private final MediaAdapter adapter;

    @Override
    public void play(String audioType, String fileName) {
        try {
            adapter.play(audioType, fileName);
        } catch (UnsupportedFormatException e) {
            System.err.println("Format not supported: " + audioType);
        } catch (FileNotFoundException e) {
            System.err.println("File not found: " + fileName);
        } catch (Exception e) {
            System.err.println("Playback error: " + e.getMessage());
        }
    }
}
```

2. Validation

java

```
class ValidatingAdapter implements PaymentProcessor {
    private final PaymentAdapter adapter;

    @Override
    public PaymentResult processPayment(double amount, String currency) {
        validateAmount(amount);
        validateCurrency(currency);
        return adapter.processPayment(amount, currency);
    }

    private void validateAmount(double amount) {
        if (amount <= 0) {
            throw new IllegalArgumentException("Amount must be positive");
        }
    }

    private void validateCurrency(String currency) {
        if (!isValidCurrency(currency)) {
            throw new IllegalArgumentException("Invalid currency: " + currency);
        }
    }
}
```

3. Logging & Monitoring

java

```java
class LoggingAdapter implements MediaPlayer {
    private final MediaAdapter adapter;
    private final Logger logger;

    @Override
    public void play(String audioType, String fileName) {
        long startTime = System.currentTimeMillis();
        try {
            logger.info("Starting playback: {} - {}", audioType, fileName);
            adapter.play(audioType, fileName);
            logger.info("Playback completed in {} ms",
                    System.currentTimeMillis() - startTime);
        } catch (Exception e) {
            logger.error("Playback failed: {}", e.getMessage());
            throw e;
        }
    }
}
```

4. Configuration Management

java

```
class ConfigurableAdapter implements MediaPlayer {
    private final Map<String, MediaAdapter> adapters;

    public ConfigurableAdapter(Properties config) {
        adapters = new HashMap<>();
        loadAdapters(config);
    }

    private void loadAdapters(Properties config) {
        config.forEach((key, value) -> {
            String format = key.toString();
            String adapterClass = value.toString();
            try {
                MediaAdapter adapter = (MediaAdapter) Class.forName(adapterClass)
                                                .newInstance();
                adapters.put(format, adapter);
            } catch (Exception e) {
                System.err.println("Failed to load adapter for " + format);
            }
        });
    }
}
```

🔗 Related Patterns

1. Adapter + Factory

java

```
class AdapterFactory {
    public static MediaPlayer createPlayer(String type) {
        switch (type) {
            case "advanced": return new MediaAdapter("vlc");
            case "legacy": return new LegacyAdapter();
            default: return new DefaultPlayer();
        }
    }
}
```

2. Adapter + Decorator

java

```
class DecoratedAdapter extends MediaAdapter {
    @Override
    public void play(String audioType, String fileName) {
        addMetadata(fileName);
        super.play(audioType, fileName);
        logPlayback(fileName);
    }
}
```

3. **Adapter + Proxy**

```
java

class ProxyAdapter implements MediaPlayer {
    private MediaAdapter realAdapter;
    private final String audioType;

    @Override
    public void play(String audioType, String fileName) {
        if (realAdapter == null) {
            realAdapter = new MediaAdapter(audioType);
        }
        realAdapter.play(audioType, fileName);
    }
}
```

🏁 Conclusion

The Adapter Pattern is a **powerful structural pattern** that makes **incompatible interfaces compatible**. This pattern is especially useful when you need to:

- **Integrate legacy systems**
- **Use third-party libraries**
- **Provide uniform interfaces** for different APIs

Key Takeaway: The Adapter pattern provides a **"translation layer"** that helps old and new code work together seamlessly.

Real-World Analogy: Just like you use a translator when traveling to a foreign country, the Adapter pattern acts as a translator in your code!

Remember: Don't overuse the pattern - if direct modification is safe and possible, choose the simpler approach.

Happy Coding! 🚀

"Bridge the gap between old and new, make incompatible systems work as one crew!"