Adapter

Design

Pattern

Adapter Design Pattern

1. Definition

The **Adapter Design Pattern** is a **structural pattern** that allows objects with incompatible interfaces to work together by acting as a **bridge** between them.
It translates one interface into another that a client expects.

Analogy:
Think of a **travel power adapter** — it converts the shape and voltage of a plug so your device can work in another country.

2. Intent / Purpose

- To enable collaboration between classes or systems that otherwise could not work together due to **interface incompatibility**.
- To **reuse** existing code without modifying it.
- To promote **flexibility** and **loose coupling**.

3. Real-World Examples

- **Power adapter**: Converts plug type from one country to another.
- **Card reader**: Converts data from memory card into a USB interface.
- **Language translator**: Converts a speaker's language to one understood by the audience.
- **Java I/O Streams**: InputStreamReader adapts a byte stream to a character stream.

4. Types of Adapters

1. **Class Adapter (Using Inheritance)**
    - Adapter inherits from both the target interface and adaptee class.
    - Uses **multiple inheritance** (possible in languages like C++, but in Java achieved by extending a class and implementing an interface).
    - Less flexible (tightly bound to adaptee).
2. **Object Adapter (Using Composition)**
    - Adapter contains an instance of adaptee and delegates calls.
    - More flexible (can adapt multiple adaptees at runtime).
    - Preferred in Java.

5. Components

1. **Target**
    - The interface expected by the client.
    - Example: WeightMachineAdapter
2. **Adaptee**
    - The existing class with a different interface that needs adapting.
    - Example: WeightMachine (returns weight in KG)
3. **Adapter**
    - Implements the Target interface and internally uses the Adaptee to fulfill the request.
    - Example: WeightMachineAdapterImpl (converts KG to Pounds)
4. **Client**
    - Works with the Target interface without knowing about the Adaptee.

6. UML Structure

Client --> Target(interface) <-- Adapter --> Adaptee

8. When to Use

- You want to **reuse existing classes** but their interface doesn't match your needs.
- You want to create a **middle layer** between old and new systems.
- You need to **integrate third-party libraries** into your project without changing their code.

9. Advantages

✅ Promotes **code reusability**.
✅ Achieves **loose coupling** between systems.
✅ Makes incompatible interfaces work together.
✅ Easy to **extend** adapters to support new conversions.

10. Disadvantages

⚠ Can increase complexity if overused.
⚠ In case of **class adapters**, you are bound by inheritance limitations.
⚠ Might hide the actual complexity behind the adapter.

---

11. Difference Between Adapter and Related Patterns

Pattern	Purpose
Adapter	Converts one interface to another.
Decorator	Adds responsibilities without changing interface.
Facade	Provides a simplified interface to a complex subsystem.
Bridge	Separates abstraction from implementation to vary them independently.

---

12. Key Interview Notes

- In Java, **object adapter** is preferred over **class adapter**.
- Adapter pattern is about **interface compatibility**, not adding features.
- Can be implemented at **compile-time** (static) or **runtime** (dynamic via reflection).
- Commonly used when integrating **legacy systems** with **modern APIs**.
