CHAPTER - 8 ADAPTER PATTERN

 **Intent**

* Convert the interface of one class (the **Adaptee**) into another interface (the **Target**) that clients expect.
* Let incompatible classes work together by “wrapping” the Adaptee.

 **Participants**

* **Target**: the interface that the client expects.
* **Adaptee**: the existing class with an incompatible interface.
* **Adapter**: implements Target, holds or extends Adaptee, and translates calls.
* **Client**: uses the Target interface.

 **Structure**

* **Object Adapter (composition)**

class Adapter implements Target {

private final Adaptee adaptee;

public Adapter(Adaptee a) { this.adaptee = a; }

@Override

public void targetMethod(...) {

// translate and delegate

adaptee.specificRequest(...);

}

}

* **Class Adapter (inheritance)**

class Adapter extends Adaptee implements Target {

@Override

public void targetMethod(...) {

// can call super.specificRequest()

}

}

 **Variants**

* **Class Adapter**
  + Uses inheritance.
  + Inherits all public/protected methods automatically.
  + Cannot adapt subclasses without writing new adapter subclasses.
* **Object Adapter**
  + Uses composition.
  + One adapter can wrap any Adaptee subclass without change.
  + Requires a field + delegation code for each interface method.
* **Two-Way Adapter**
  + Implements both interfaces to translate calls in both directions (rare).

 **Key Differences vs. Decorator**

| **Aspect** | **Adapter** | **Decorator** |
| --- | --- | --- |
| **Intent** | Change interface so two classes work | Add responsibilities to an object dynamically |
| **Interface** | Adapter returns a **different** interface | Decorator returns the **same** interface |
| **Use case** | Integrate legacy or third-party APIs | Layer on features (e.g. caching, logging) |

 **When to Use**

* **Adapter**: when you need to make two incompatible interfaces work together without modifying either.
* **Decorator**: when you need to add or override behavior on an object while keeping its interface intact.

 **Composition – Object Adapter**

* Pros: maximal flexibility. One adapter class can wrap any future subclass without change.
* Cons: one extra object reference per adapter and manual delegation for each method you want to expose.
* Adapter will have a Adaptee class so it will be enough to wrap any subclass of adaptee.

 **Class (inheritance) adapter**

* Pros: zero delegation boilerplate for inherited methods and one object instead of two.
* Cons: if the adaptee hierarchy grows, you must subclass your adapter to handle each new variant.
*  Because it subclasses the Adaptee directly, it automatically inherits all of its public and protected methods without writing delegate methods.
*  **But** if someone later creates AdvancedAdaptee extends Adaptee that adds new operations, your existing adapter won’t expose those new methods. You’d need to create a new adapter subclass (e.g. AdvancedAdapteeAdapter extends AdvancedAdaptee implements Target) to handle the added functionality.

**Adapter vs Facade Pattern:**

The intent of the Adapter Pattern is to **alter** an interface so that it matches one a client is expecting. Theintent of the Facade Pattern is to provide a **simplified** interface to a subsystem.

**Facade Pattern**  
Provides a simple, high‐level interface to a complex subsystem of classes, making it easier for clients to interact without knowing the details of the subsystem’s components.

**1. Intent**

* **Simplify** interaction with a complex set of classes.
* **Decouple** clients from subsystem implementations.

**2. Participants**

* **Facade**
  + Defines a high‐level interface (doWork(), startProcess(), etc.).
  + Delegates client requests to the appropriate subsystem classes.
* **Subsystem classes**
  + Implement the actual work.
  + Have no knowledge of the Facade.
* **Client**
  + Calls methods on the Facade, unaware of the subsystem’s internal structure.

**3. Structure**

Client → Facade → SubsystemA

SubsystemB

SubsystemC

* **Facade** holds references to each **Subsystem** object.
* **Client** only knows and uses **Facade** methods.

**4. Typical Usage**

// Subsystem classes

class AuthService { void login(); void logout(); }

class DataService { void fetchData(); void saveData(); }

class NotificationService { void notifyUser(); }

// Facade

class ApplicationFacade {

private final AuthService auth = new AuthService();

private final DataService data = new DataService();

private final NotificationService notify = new NotificationService();

public void startApp() {

auth.login();

data.fetchData();

notify.notifyUser();

}

public void shutdown() {

data.saveData();

auth.logout();

}

}

// Client

public class Main {

public static void main(String[] args) {

ApplicationFacade app = new ApplicationFacade();

app.startApp();

// … user works …

app.shutdown();

}

}

**5. Benefits**

* **Reduces learning curve**: Clients only see the Facade’s simple API.
* **Loosely couples**: Clients remain unaware of subsystem classes.
* **Easier maintenance**: Subsystem internals can change without impacting clients.

**6. Drawbacks**

* **Leaky abstraction**: If the Facade doesn’t expose a needed operation, clients may bypass it and call subsystem classes directly—creating tight coupling.
* **Over‐simplification**: Too many responsibilities in one Facade can lead to a “God object.”

**7. When to Use**

* When you have a **complex subsystem** with many interdependent classes.
* When you want to provide a **simple entry point** for common tasks.
* To **layer** your architecture: each layer exposes a Facade to the layer above.

**Law of Demeter** (a.k.a. “Don’t talk to strangers”), and it’s all about keeping each class’s “circle of friends” small so you’re not tightly coupled to lots of internal pieces. Here’s what it’s saying, step by step:

**1. Who you’re allowed to talk to**

From *inside* any method of some object **A**, you may only call methods on:

1. **A itself**
2. **Objects passed in as parameters** to the method
3. **Objects you create** inside the method
4. **Fields (components) of A**—things A already holds in its instance variables

You **must not** call methods on objects you got back from those things. No chaining.

**2. Why avoid chained calls**

Imagine you write:

// station → thermometer → temperature

float temp = station.getThermometer() // 1st call

.getTemperature(); // 2nd call

* Your code now “knows” about **Station** and **Thermometer**.
* If Thermometer’s API changes, or if Station stops exposing a Thermometer directly, your code breaks.

**3. How to fix it (delegate through Station)**

Instead, give Station the job of talking to its Thermometer:

// inside Station class

public float getTemperature() {

return this.thermometer.getTemperature();

}

// then client code simply does

float temp = station.getTemperature();

* Now your client only knows about **Station**.
* If Station’s internals change (maybe it swaps to a Sensor instead of a Thermometer), your client code is untouched.

**4. “Circle of friends” metaphor**

* **Friends you can call directly**:
  + Yourself
  + People (objects) you invited in (method parameters)
  + People you just hired (objects you created)
  + People you already live with (your fields)
* **Strangers you mustn’t call directly**:
  + Their friends—i.e., objects returned from your friends.

When you need something from a stranger’s friend, ask *your* friend to go get it for you.

**Bottom line**

The Law of Demeter helps you write code that’s easier to maintain and less fragile by **avoiding deep chains of object calls**. Whenever you find yourself doing a.getB().getC().doSomething(), think about adding a doSomething() or getC() method to **B** (or even to **A**), so callers only ever talk to their immediate friends.