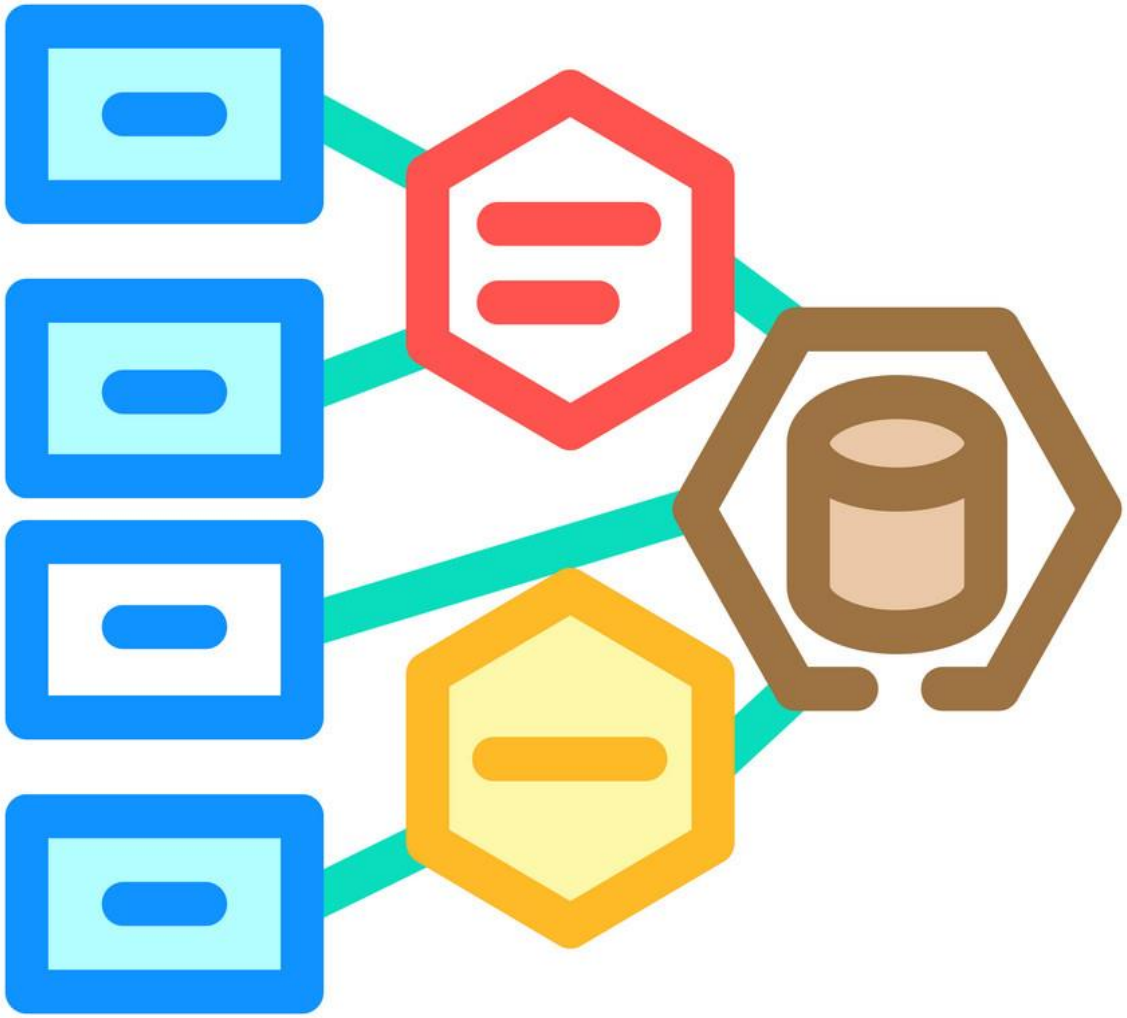
Software Design and architecture

Fall 2024

Dr. Natalia Chaudhry



Outline

- Object-Oriented Design with UML
- Elementary design patterns

Basics of Object-Oriented Programming

- Objects, classes
- Static members
- Constructors
- Abstract classes and interfaces
- Association (aggregation vs composition)
- Inheritance
- Polymorphism and static vs dynamic binding
- Recall OOP

Language Features for Object-Oriented Implementation

- Packages One major theme in object-oriented paradigm is reuse.
- This decreases development time, reduces code size, and increases reliability.
- The Java language comes with a large number of classes (numbering in the thousands) that can be used for a variety of uses: networking, GUI, database management, and so on. We will use some classes from Java quite extensively so that we can focus more on the design issues.
- This is also consistent with the theme of reuse.
- The Java classes are spread over what are called packages

- A package is a collection of classes. It is usually named as a sequence of lowercase letters and periods.
- Some of the major packages are **java.lang,java.util, java.awt, javax.swing, java.io,** and **java.lang.reflect**. The package **java.lang** contains classes and interfaces that are fundamental to the language

To import all of the members of a package

```
import java.util.*;
```

Introduction to Object-Oriented Analysis, Design, Implementation and Refactoring

Elementary Design Patterns

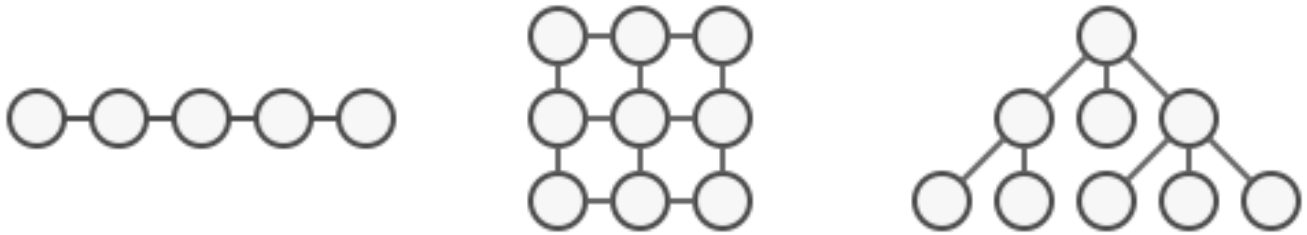
- Working on a variety of projects, a software engineer gets exposure to problems that are common to multiple scenarios, which hones his/her ability to identify repeated instances of problems and spell out solutions for them fairly quickly.
- From an objectoriented perspective, what it means is that two different applications may provide design issues that are alike; the solutions may involve the development of a set of classes with similar functionalities and relationships.
- Thus the class structures for the two subproblems may end up being the same although there may be differences in details

- Consider two applications, one a university course registration system and the other a human resource (HR) system for some organisation.
- Although the applications are quite different, the scenarios have similarity: both involve reading information which is data related to some application from disk and then sorting the data based on some fields in it.
- An efficient sorting mechanism should be used in both cases.
- The task in object-oriented systems is to recognise the necessary classes, interfaces and relationships between them for solving a specific design problem. Such an approach, which can then be tailored to solve similar **design problems** that recur in a multitude of applications, is called a **design pattern**.
- A pattern is a way of doing something. Design patterns are partial solutions to common problems such as separating an interface from a number of alternate implementations

Iterator

- In many applications we need to maintain collections which are objects that store other objects.
- For example, a telephone company system could have a collection object that stores an object for each of its customers; an airline system is likely to maintain information about each of its flights and the references to them may be stored in a collection object.
- Depending on the type of application, the actual data structure employed may differ
- Popular data structures that implement collections include linked lists, queues, stacks, double-ended queues, binary search trees, B-Trees and hash tables.

Collections are one of the most used data types in programming. Nonetheless, a collection is just a container for a group of objects.



Various types of collections.

Most collections store their elements in simple lists. However, some of them are based on stacks, trees, graphs and other complex data structures.
But no matter how a collection is structured, it must provide some way of accessing its elements so that other code can use these elements. There should be a way to go through each element of the collection without accessing the same elements over and over.

Adding more and more traversal algorithms to the collection gradually blurs its primary responsibility, which is efficient data storage.

Additionally, some algorithms might be tailored for a specific application, so including them into a generic collection class would be weird.

The main idea of the Iterator pattern is to extract the traversal behavior of a collection into a separate object called an *iterator*.

Iterator is a behavioral design pattern that lets you traverse elements of a collection without exposing its underlying representation (list, stack, tree, etc.).

- In addition to implementing the algorithm itself, an iterator object encapsulates all of the traversal details, such as the current position and how many elements are left till the end.
- Because of this, several iterators can go through the same collection at the same time, independently of each other.

- All iterators must implement the same interface.
- This makes the client code compatible with any collection type or any traversal algorithm as long as there's a proper iterator.
- If you need a special way to traverse a collection, you just create a new iterator class, without having to change the collection or the client.

The **Iterator Interface** defines the methods necessary to traverse elements of a collection. It encapsulates the traversal logic and provides a unified interface regardless of the underlying collection type (e.g., arrays, lists, trees).

Key Responsibilities:

- Provide access to elements in a collection sequentially.
- Ensure the caller does not need to understand the underlying data structure.

Common Methods:

- **hasNext()**: Returns true if there are more elements in the collection.
- **next()**: Returns the next element in the collection.
- **remove()**: (Optional) Removes the last element returned by the iterator.

```
public interface Iterator<T> {
    boolean hasNext();
    T next();
}
```

Collector Interface (Aggregate Interface)

The **Collector Interface** provides a way to create an **Iterator** for the collection it represents. It acts as a factory for the Iterator, decoupling the collection from the iteration logic.

Key Responsibilities:

•Define a method to return an **Iterator**.

•Allow iteration over the collection by providing the **Iterator** object.

Common Method:

•**createIterator()**: Returns an **Iterator** object for the collection.

```
public interface Aggregate<T> {
    Iterator<T> createIterator();
}
```

How to Implement

1.Declare the iterator interface. At the very least, it must have a method for fetching the next element from a collection. But for the sake of convenience you can add a couple of other methods, such as fetching the previous element, tracking the current position, and checking the end of the iteration.

2.Declare the collection interface and describe a method for fetching iterators. The return type should be equal to that of the iterator interface. You may declare similar methods if you plan to have several distinct groups of iterators.

3.Implement concrete iterator classes for the collections that you want to be traversable with iterators. An iterator object must be linked with a single collection instance. Usually, this link is established via the iterator's constructor.

4. Implement the collection interface in your collection classes. The main idea is to provide the client with a shortcut for creating iterators, tailored for a particular collection class. The collection object must pass itself to the iterator's constructor to establish a link between them.
5. Go over the client code to replace all of the collection traversal code with the use of iterators. The client fetches a new iterator object each time it needs to iterate over the collection elements.

Suppose we have a collection of book titles, and we want to traverse them using the Iterator pattern.

Step 1: Define the Iterator Interface

This interface will allow us to iterate through the collection.

```
public interface Iterator<T> {
    boolean hasNext(); // Check if there are more elements
    T next();          // Retrieve the next element
}
```

Step 2: Define the Collector Interface (Aggregate Interface)

This interface provides a method to create an iterator for the collection.

```
public interface Aggregate<T> {
    Iterator<T> createIterator(); // Create an iterator for the collection
}
```

Step 3: Implement the Collection (Aggregate)

The collection will implement the **Aggregate Interface** and provide a concrete iterator.

```
import java.util.ArrayList;
import java.util.List;

public class BookCollection implements
Aggregate<String> {
    private final List<String> books = new
ArrayList<>();

    // Add a book to the collection
    public void addBook(String book) {
        books.add(book);
    }

    // Create an iterator for the collection
    @Override
    public Iterator<String> createIterator() {
        return new BookIterator();
    }

    // Inner class for the concrete iterator
    private class BookIterator implements Iterator<String> {
        private int index = 0;

        @Override
        public boolean hasNext() {
            return index < books.size(); // Check if more elements
exist
        }
```

```
        @Override
        public String next() {
            if (hasNext()) {
                return books.get(index++); // Return the current element and move to the next
            }
            return null;
        }
    }
}
```

Step 4: Use the Iterator Pattern in Client Code

The client code uses the iterator to access elements without needing to know the details of the collection.

```
public class IteratorPatternExample {
    public static void main(String[] args) {
        // Create the book collection
        BookCollection bookCollection = new BookCollection();
        bookCollection.addBook("The Great Gatsby");
        bookCollection.addBook("To Kill a Mockingbird");
        bookCollection.addBook("1984");
        bookCollection.addBook("Pride and Prejudice");

        // Get the iterator
        Iterator<String> iterator = bookCollection.createIterator();

        // Traverse the collection using the iterator
        System.out.println("Books in the collection:");
        while (iterator.hasNext()) {
            System.out.println(iterator.next());
        }
    }
}
```

Pros

• *Single Responsibility Principle*. You can clean up the client code and the collections by extracting bulky traversal algorithms into separate classes.

• *Open/Closed Principle*. You can implement new types of collections and iterators and pass them to existing code without breaking anything.

• You can iterate over the same collection in parallel because each iterator object contains its own iteration state.

• For the same reason, you can delay an iteration and continue it when needed.

Cons

•Applying the pattern can be an overkill if your app only works with simple collections.

• Using an iterator may be less efficient than going through elements of some specialized collections directly.

Singleton

- in many situations we want to ensure that there is just one object of a certain class.
- For example, although a computer system may have many printers, there is usually only one spooler.
- A company has only one president.
- A single-processor system obviously can have only one CPU

- To create a class that can only be instantiated once, we note that the constructor cannot have the public access specifier.
- Instead, we provide a method called instance() that returns the only instance of the class.

```
public class B {
  private static B singleton;
  private B() {
  }
  public static B instance() {
    if (singleton == null) {
      singleton = new B();
    }
    return singleton;
  }
  // application code
}
```

The major observation to be made here is that to get the only instance of class B, a client invokes the static method instance. This is because the constructor is private, so the code from outside the class cannot instantiate B. When the class is loaded, the field singleton will be set to null. In the very first call to instance, an instance of B is created and the reference stored in singleton. Further calls to instance result in no new allocations, and the value in singleton is returned.

```
public class B {
  private static B singleton;
  private B() {
  }
  public static B instance() {
    if (singleton == null) {
      singleton = new B();
    }
    return singleton;
  }
  // application code
}
```

Subclassing a Singleton is a design decision that allows additional functionality or specialized behavior to be added to the Singleton while retaining its controlled instance management.

Customization for Polymorphism

•When polymorphic behavior is required, and the Singleton serves as a base class.

•Example: A DatabaseConnection Singleton may need to support different databases (e.g., MySQLConnection or PostgresConnection) through subclasses.

Testing and Mocking

•In testing scenarios, subclassing can be used to create mock Singletons or specialized implementations for test cases without modifying the original Singleton.

The **Singleton pattern** solves two problems at the same time, violating the ***Single Responsibility Principle***:

1. **Ensure that a class has just a single instance.** Why would anyone want to control how many instances a class has? The most common reason for this is to control access to some shared resource—for example, a database or a file.

Here's how it works: imagine that you created an object, but after a while decided to create a new one. Instead of receiving a fresh object, you'll get the one you already created.

Note that this behavior is impossible to implement with a regular constructor since a constructor call **must** always return a new object by design.

2.Provide a global access point to that instance.

Global variables are very unsafe since any code can potentially overwrite the contents of those variables and crash the app.

Just like a global variable, the Singleton pattern lets you access some object from anywhere in the program. However, it also protects that instance from being overwritten by other code.

All implementations of the Singleton have these two steps in common:

Make the default constructor private, to prevent other objects from using the new operator with the Singleton class.
Create a static creation method that acts as a constructor. Under the hood, this method calls the private constructor to create an object and saves it in a static field. All following calls to this method return the cached object.

Pros

• You can be sure that a class has only a single instance.
• You gain a global access point to that instance.
• The singleton object is initialized only when it's requested for the first time.

Cons

• Violates the *Single Responsibility Principle*. The pattern solves two problems at the time.
• The Singleton pattern can mask bad design, for instance, when the components of the program know too much about each other.
• The pattern requires special treatment in a multithreaded environment so that multiple threads won't create a singleton object several times.
• It may be difficult to unit test the client code of the Singleton because many test frameworks rely on inheritance when producing mock objects. Since the constructor of the singleton class is private and overriding static methods is impossible in most languages, you will need to think of a creative way to mock the singleton. Or just don't write the tests. Or don't use the Singleton pattern.

Adapter

Adapter is a structural design pattern that allows objects with incompatible interfaces to collaborate.

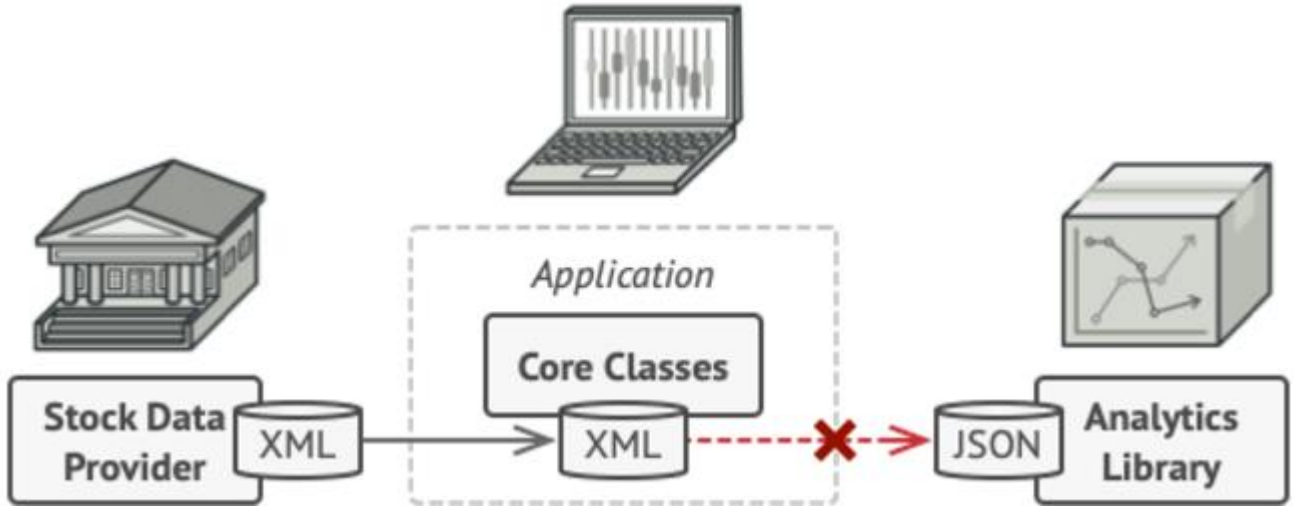
It acts as a bridge between the two interfaces by wrapping one interface and translating it into another.

Real-World Analogy

Imagine you have a phone charger plug (adaptee) that only fits UK sockets, but you are in the US. An **adapter** converts the plug so it fits into a US socket, enabling the charger to work without changing its internal structure.

Imagine that you're creating a stock market monitoring app. The app downloads the stock data from multiple sources in XML format and then displays nice-looking charts and diagrams for the user.

At some point, you decide to improve the app by integrating a smart 3rd-party analytics library. But there's a catch: the analytics library only works with data in JSON format.



You can't use the analytics library "as is" because it expects the data in a format that's incompatible with your app.

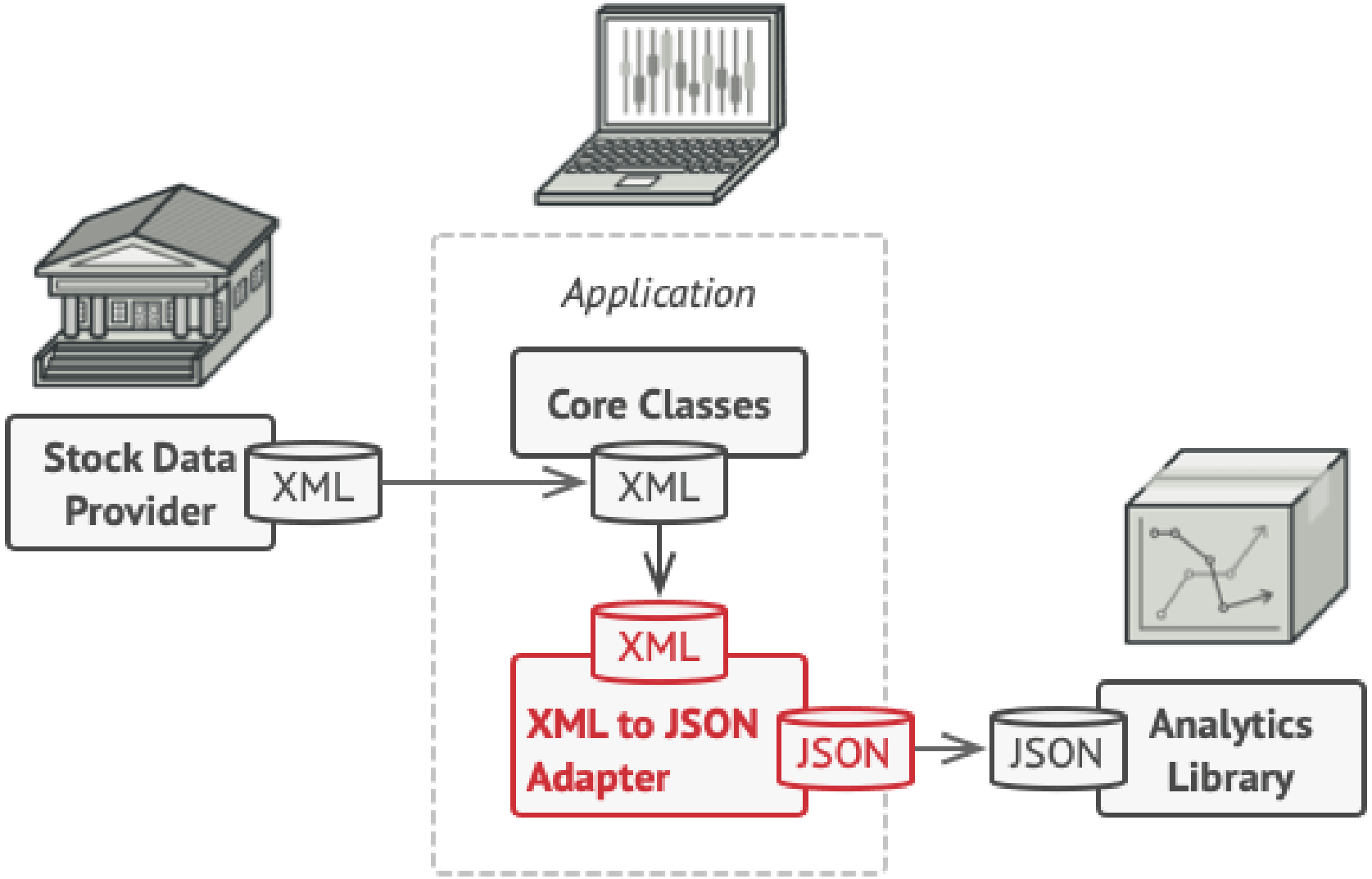
You could change the library to work with XML. However, this might break some existing code that relies on the library.

And worse, you might not have access to the library's source code in the first place, making this approach impossible.

You can create an *adapter*. This is a special object that converts the interface of one object so that another object can understand it.

Adapters can not only convert data into various formats but can also help objects with different interfaces collaborate. Here's how it works:

1.The adapter gets an interface, compatible with one of the existing objects.
2.Using this interface, the existing object can safely call the adapter's methods.
3.Upon receiving a call, the adapter passes the request to the second object, but in a format and order that the second object expects.


Application
Core Classes
Stock Data Provider
XML
XML
XML
XML to JSON Adapter
JSON
JSON
Analytics Library

- To solve the dilemma of incompatible formats, you can create XML-to-JSON adapters for every class of the analytics library that your code works with directly.
- Then you adjust your code to communicate with the library only via these adapters.
- When an adapter receives a call, it translates the incoming XML data into a JSON structure and passes the call to the appropriate methods of a wrapped analytics object.

Key Components of the Adapter Pattern:

1. **Target Interface**: Defines the domain-specific interface used by the client code.
2. **Adaptee**: The existing class or component that needs to be adapted.
3. **Adapter**: The class that implements the target interface and translates requests from the client into a format the Adaptee can understand.
4. **Client**: The code that interacts with the Target Interface.

Imagine you are integrating an old payment processing system (Adaptee) with a new e-commerce platform (Client) that expects a modern interface (Target).

Target

```
interface PaymentProcessor {
    void processPayment(double amount);
}
```

Adaptee

```
class OldPaymentSystem {
    public void makePayment(double
amount) {
        System.out.println("Payment of $" +
amount + " processed using the old
system.");
    }
}
```

Adapter

```
class PaymentAdapter implements
PaymentProcessor {
    private OldPaymentSystem oldPaymentSystem;

    public PaymentAdapter(OldPaymentSystem
oldPaymentSystem) {
        this.oldPaymentSystem = oldPaymentSystem;
    }
    @Override
    public void processPayment(double amount) {
        // Translate the call to the Adaptee's method
        oldPaymentSystem.makePayment(amount);
    }
}
```

Client

```
public class ECommerceApp {
    public static void main(String[] args) {
        OldPaymentSystem oldPaymentSystem = new OldPaymentSystem();
        PaymentProcessor adapter = new PaymentAdapter(oldPaymentSystem);

        // Client uses the new interface
        adapter.processPayment(250.75);
    }
}
```

The **Adapter Pattern** helps in integrating legacy systems into new ones without modifying the original code.

- It allows classes with incompatible interfaces to collaborate.
- It adheres to the **Open/Closed Principle** by extending functionality through the adapter without changing existing code.

This pattern is widely used in:

- Wrapping libraries or APIs with different interfaces.
- Building drivers for hardware components.
- Translating protocols or data formats.

Pros

• *Single Responsibility Principle*. You can separate the interface or data conversion code from the primary business logic of the program.

• *Open/Closed Principle*. You can introduce new types of adapters into the program without breaking the existing client code, as long as they work with the adapters through the client interface.

Cons

• The overall complexity of the code increases because you need to introduce a set of new interfaces and classes. Sometimes it’s simpler just to change the service class so that it matches the rest

That’s it