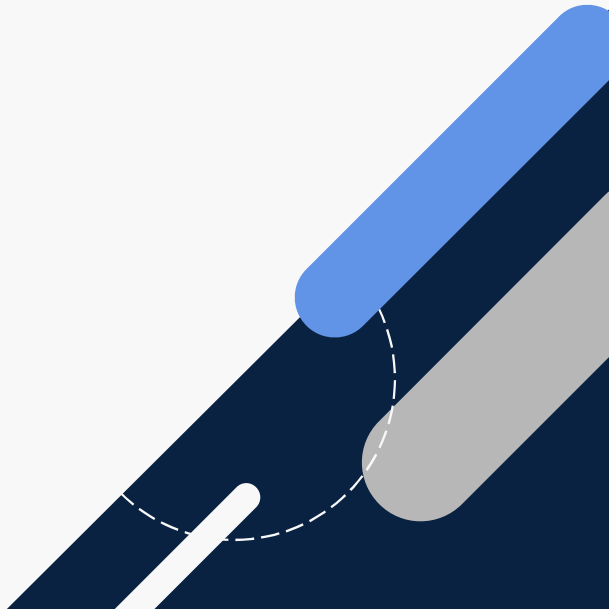
Pattern Adapter



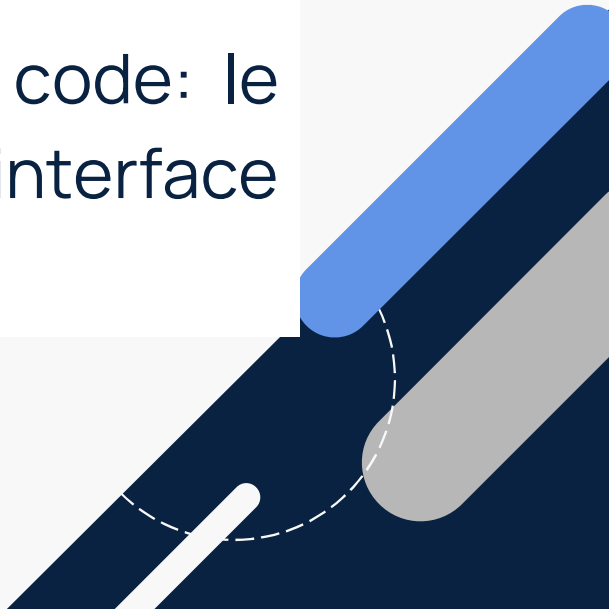
Définition

L’ Adaptateur :

- Est un patron de conception structurel qui permet de faire collaborer des objets ayant des interfaces incompatibles.
- Fonctionne comme un pont entre deux interfaces incompatibles.
- Permet de combiner la capacité de deux interfaces indépendantes.

Objectif

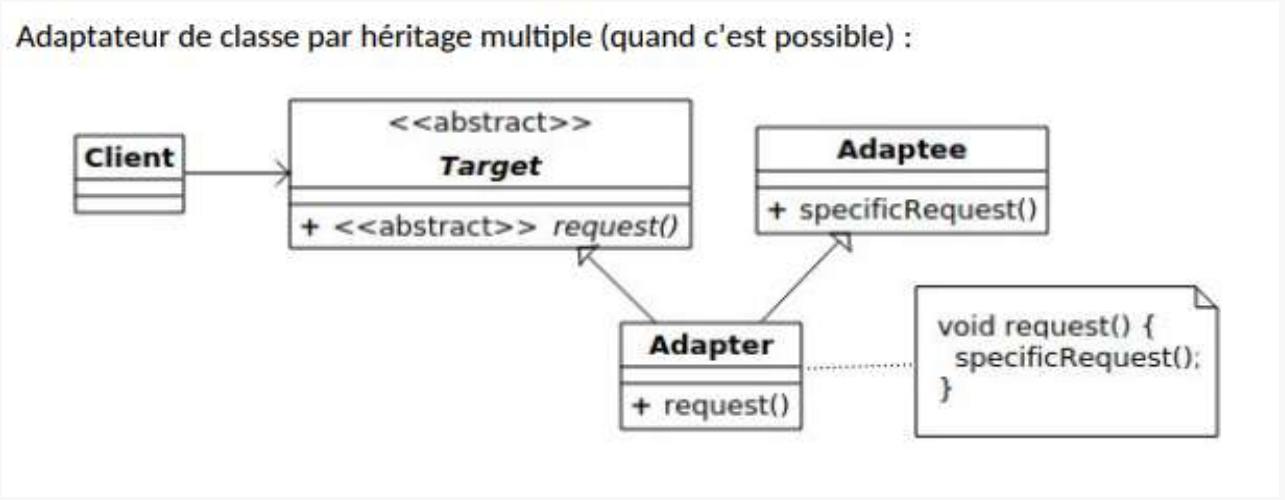
- Convertir l'interface d'une classe en une autre interface attendue par le client (interface cible) afin de permettre à des classes incompatibles d'interagir ensemble.
- Souvent motivé par la réutilisation de code: le code réutilisé doit être conforme à une interface requise



Structure UML

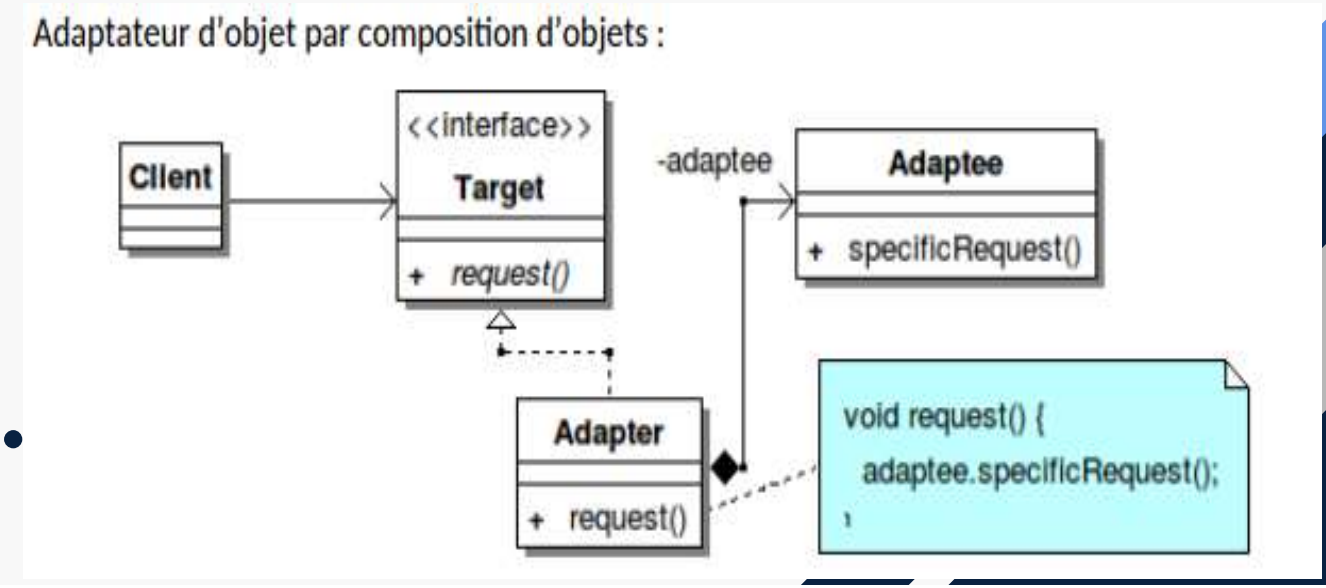
Adaptateur de classe:

par héritage multiple/interfaces



Adaptateur d'objet:

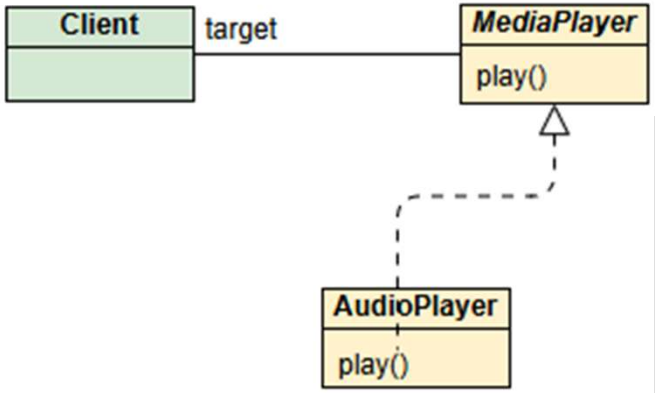
par composition d'objets



Exemple d'implémentation

On dispose d'une interface MediaPlayer et une classe concrète AudioPlayer implémentant l'interface MediaPlayer.

AudioPlayer peut lire les fichiers audio au format mp3 par défaut.




```
class AudioPlayer implements MediaPlayer{
public void play(String filename) {
 System.out.println("Lecture de MP3 " + filename + " via AudioPlayer...");  }
}
```

Exemple d’implémentation

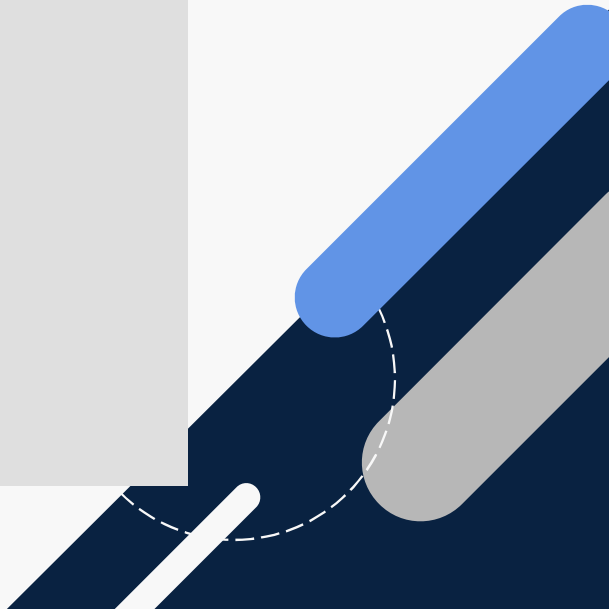
Problème:

Nous voulons faire en sorte qu'AudioPlayer puisse également lire d'autres formats vlc et mp4.

Les méthodes ne sont ni

- homogènes
- interchangeables

```
class Mp4Player {
   public void playMp4(String filename) { ... }
}
class VlcPlayer {
   public void playVlc(String filename) { ... }
}
```

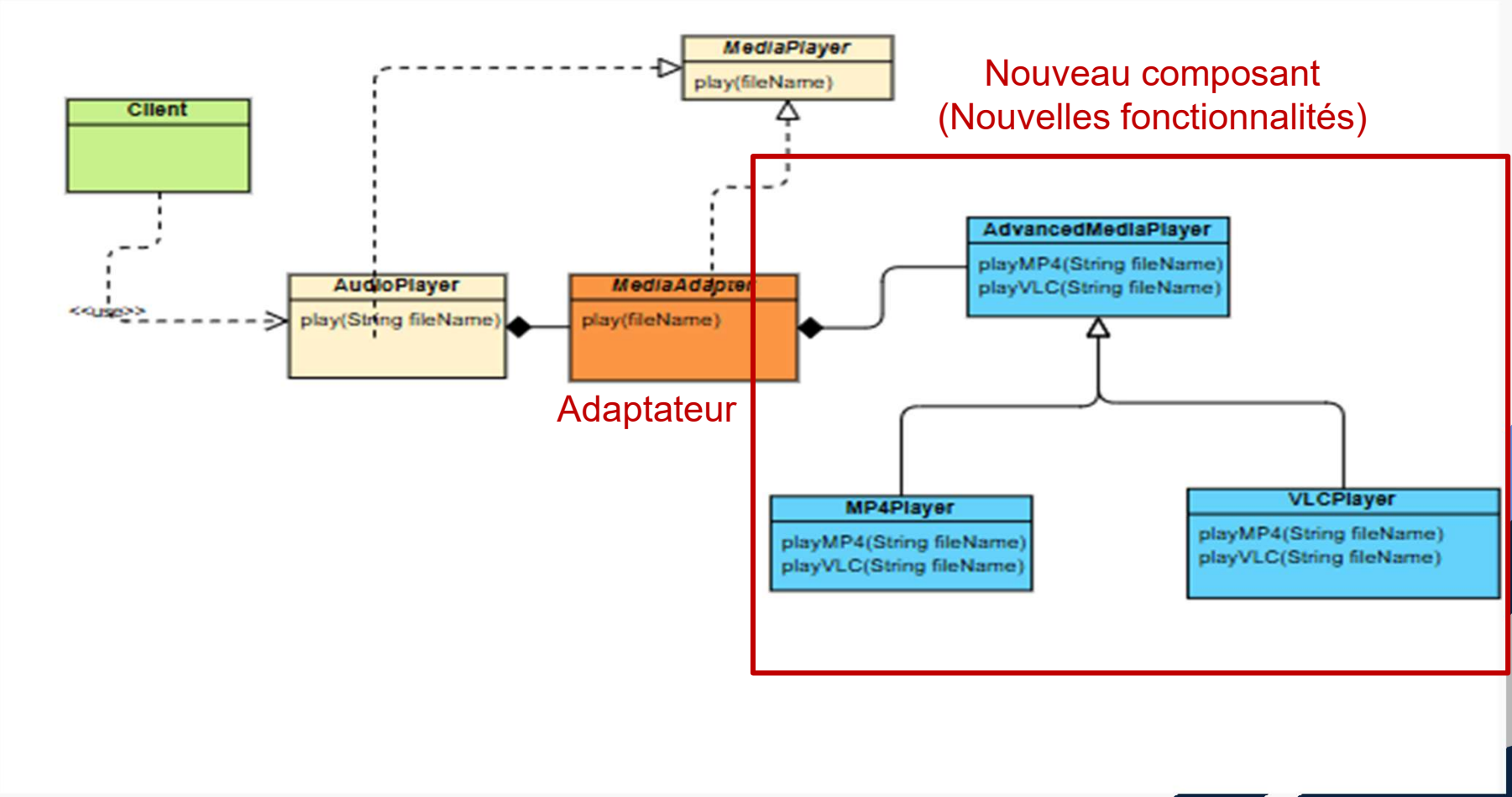


Exemple d’implémentation

Solution:

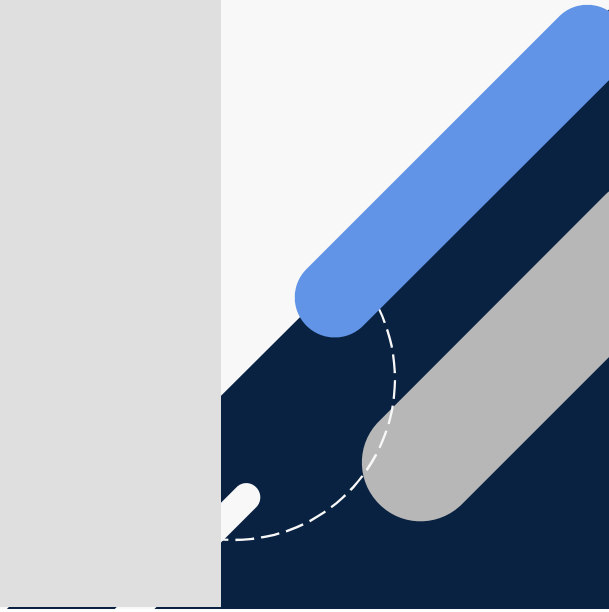
- On intègre à la solution existante un autre composant: interface AdvancedMediaPlayer et des classes concrètes MP4Player et VLCPlayer implémentant l'interface AdvancedMediaPlayer qui permettent de lire des fichiers aux formats mp4 et vlc, respectivement.
- On doit créer aussi une classe d'adaptateur MediaAdapter qui implémente l'interface MediaPlayer et utilise des objets AdvancedMediaPlayer pour lire le format requis →Adaptateur d’objets (composition)

Exemple d'implémentation



MediaAdapter

```
public class MediaAdapter implements MediaPlayer {
    private AdvancedMediaPlayer advancedPlayer;

    public MediaAdapter(String extension) {
        if (extension.equalsIgnoreCase("vlc")) {
            advancedPlayer = new VlcPlayer();
        } else if (extension.equalsIgnoreCase("mp4")) {
            advancedPlayer = new Mp4Player();
        }
    }

    @Override
    public void play(String filename) {
        String extension = getExtension(filename);

        if (extension.equalsIgnoreCase("vlc")) {
            advancedPlayer.playVlc(filename);
        } else if (extension.equalsIgnoreCase("mp4")) {
            advancedPlayer.playMp4(filename);
        }     }
```

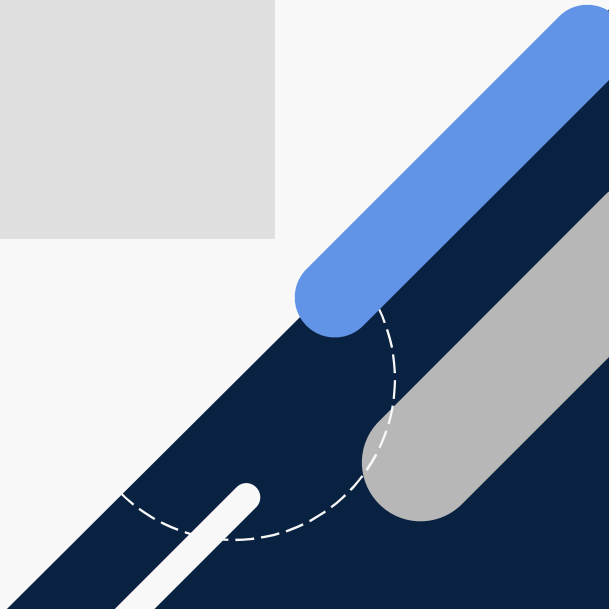


AudiPlayer

```
public class AudioPlayer implements MediaPlayer {

    private MediaAdapter adapter;

    @Override
    public void play(String filename) {
        String extension = getExtension(filename);

        if (extension.equalsIgnoreCase("mp3")) {
            System.out.println("Lecture du MP3 : " + filename);
        } else if (extension.equalsIgnoreCase("vlc") || extension.equalsIgnoreCase("mp4")) {
            adapter = new MediaAdapter(extension);
            adapter.play(filename);
        } else {
            System.out.println("Format non supporté : " + extension);
        }
    }
```

Le client (programme principal)

```
public class Main {
    public static void main(String[] args) {
        MediaPlayer player = new AudioPlayer();

        player.play("musique.mp3");
        player.play("clip.mp4");
        player.play("film.vlc");
        player.play("document.avi");
    }
}
```



Avantages des adaptateurs

Avantage	Explication
Découplage total	Le client ne dépend pas des classes concrètes (`Mp4Player`, `VlcPlayer`), juste de `MediaPlayer`.
Extensibilité	Il est possible d'ajouter un `AviPlayer` ou `FlacPlayer` sans changer le code client.
Encapsulation	Le client ignore tout de la logique interne d'adaptation (création de `MediaAdapter`).
Substitution	Il est possible de remplacer `AudioPlayer` par une autre implémentation (`SmartPlayer`, `StreamingPlayer`) sans casser le code du client.

Problématique et solution des registres dynamiques

- L'utilisation de if/else: rend le système rigide (OCP pas vraiment respecté)

⇒ Solution: utiliser un **registre dynamique** des adaptateurs afin de permettre d'enregistrer dynamiquement de nouveaux formats multimédias, sans modifier le code existant.

⇒ Principe:

- AudioPlayer n'a plus besoin de connaître les types (mp4, vlc, etc.).
- Les adaptateurs sont déclarés dans un registre (une Map) au démarrage.
- Chaque adaptateur est créé à la volée avec une fabrique (Supplier<AdvancedMediaPlayer>).

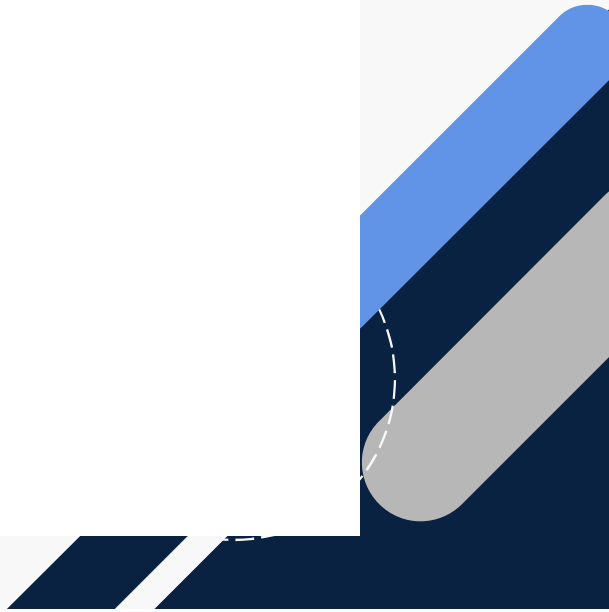
L’interface fonctionnelle Supplier

Qu’est-ce que Supplier<T> ?

- Supplier<T> est une interface fonctionnelle du package java.util.function introduite avec Java 8.
- Elle représente une “fabrique” d’objets (une fonction sans argument qui retourne une valeur de type T).

Définition :

```
@FunctionalInterface
public interface Supplier<T> {
T get(); // produit une instance de T
}
```



MediaAdapter avec registre dynamique

```
import java.util.Map;
import java.util.function.Supplier;

public class MediaAdapter implements MediaPlayer {

    // Registre dynamique d'adaptateurs => map immuable (non modifiable)
    private static final Map<String, Supplier<AdvancedMediaPlayer>> registry = Map.of(
        "vlc", VlcPlayer::new, // Appel constructeur => new VlcPlayer()
        "mp4", Mp4Player::new // Appel constructeur => new Mp4Player()
    );

    private AdvancedMediaPlayer advancedPlayer;

    public MediaAdapter(String extension) {
        Supplier<AdvancedMediaPlayer> supplier = registry.get(extension.toLowerCase());
        if (supplier != null) {
            this.advancedPlayer = supplier.get();
        }
    }
```

MediaAdapter avec registre dynamique

```
@Override
public void play(String filename) {
    if (advancedPlayer == null) {
        System.out.println("Aucun adaptateur disponible pour : " + filename);
        return;
    }
    String extension = getExtension(filename);
    if (advancedPlayer instanceof VlcPlayer) {
        advancedPlayer.playVlc(filename);
    } else if (advancedPlayer instanceof Mp4Player) {
        advancedPlayer.playMp4(filename);
    } else {
        System.out.println("Format non reconnu : " + extension);
    }
}
private String getExtension(String filename) {
    int i = filename.lastIndexOf('.');
    return (i > 0) ? filename.substring(i + 1) : "";
} }
```

AudioPlayer

```
public class AudioPlayer implements MediaPlayer {

    @Override
    public void play(String filename) {
        String extension = getExtension(filename);

        if (extension.equalsIgnoreCase("mp3")) {
            System.out.println("Lecture du MP3 : " + filename);
        } else {
            MediaAdapter adapter = new MediaAdapter(extension);
            adapter.play(filename);
        }
    }

    private String getExtension(String filename) {
        int i = filename.lastIndexOf('.');
        return (i > 0) ? filename.substring(i + 1) : "";
    }
}
```

Exercice : Convertisseur de messages

Une entreprise développe une application de messagerie interne. Elle veut intégrer un système de notification externe qui envoie des messages via différents canaux :

- Email
- SMS
- Slack (chat d'équipe)

Problème :le système existant appelle toujours une méthode send(String message),mais les classes des nouveaux canaux n'ont pas la même interface.

Exercice : Convertisseur de messages

Soit l'interface du système existant :

```
public interface MessageSender {    void send(String message);}
```

Et soit les nouvelles classes à intégrer:

```
public class EmailService {
public void sendEmail(String subject, String body) {
System.out.println("Envoi Email : " + subject + " I " + body);    }}
public class SmsService {
public void sendSms(String phoneNumber, String text) {
System.out.println("Envoi SMS à " + phoneNumber + " : " + text);
}}
```

Exercice : Convertisseur de messages

Travail demandé:

1. Implémentez un Adapter (ou plusieurs) pour rendre ces classes compatibles avec l'interface MessageSender: un adapter pour chaque service (Email, SMS, Slack)
2. Créez un MessageSenderFactory qui choisit dynamiquement le bon adapter selon un paramètre,
3. Simuler l'envoi de messages depuis le client principal, sans changer le code client.