**Imagine a Real-World Scenario:**

Think of a news agency that delivers breaking news to subscribers. Subscribers can receive updates either by email or SMS. The agency doesn't need to know who exactly gets the news; it just sends out news updates and lets subscribers handle them.

**Publisher Interface (INewsPublisher):**

* The news agency wants to communicate with its subscribers without knowing the specifics of how they receive news.
* So, it defines a set of rules that publishers (news agencies) should follow to interact with subscribers.
* The rules include methods for adding subscribers, removing subscribers, and notifying subscribers.

**Concrete Publisher (NewsAgency):**

* The actual news agency implements those rules from the interface.
* It keeps track of subscribers and has methods to add and remove them.
* When there's breaking news, it sends out notifications to all subscribers.

**Subscriber Interface (INewsSubscriber):**

* Subscribers are entities that receive news updates from the agency.
* They also have a set of rules they follow, defined in an interface.
* The interface requires them to have a method that handles receiving news updates.

**Concrete Subscribers (EmailSubscriber and SmsSubscriber):**

* These are the real people or systems that get news updates.
* Each subscriber (like EmailSubscriber and SmsSubscriber) follows the rules from the subscriber interface.
* They implement the Update method that defines what they do when they receive news.

**Client Code (Main Method):**

* This is where everything comes together.
* You create a news agency, along with subscribers of different types.
* You add subscribers to the agency's list.

When there's important news, the agency notifies all its subscribers, and each subscriber reacts accordingly.

**In Simple English:**

The Observer pattern is like a news agency and its subscribers. The agency doesn't know who its subscribers are or how they receive news (email or SMS). It just sends out news to everyone who's subscribed. Subscribers, in turn, are set up to receive and handle news updates. They follow a common set of rules, so the agency can notify them without knowing the specifics of how they're notified. This pattern allows for a flexible and dynamic relationship between entities where changes in one entity are automatically reflected in others.

**Step 1: Define the Publisher Interface (INewsPublisher)**

The INewsPublisher interface defines the contract for objects that will publish news updates. It includes methods to add and remove subscribers and notify them about updates.

public interface INewsPublisher

{

void AddSubscriber(INewsSubscriber subscriber);

void RemoveSubscriber(INewsSubscriber subscriber);

void NotifySubscribers(string newsUpdate);

}

**Step 2: Implement the Concrete Publisher (NewsAgency)**

The NewsAgency class implements the INewsPublisher interface. It maintains a list of subscribers and provides methods to manage subscribers and notify them about news updates.

public class NewsAgency : INewsPublisher

{

private List<INewsSubscriber> \_subscribers = new List<INewsSubscriber>();

public void AddSubscriber(INewsSubscriber subscriber)

{

\_subscribers.Add(subscriber);

}

public void RemoveSubscriber(INewsSubscriber subscriber)

{

\_subscribers.Remove(subscriber);

}

public void NotifySubscribers(string newsUpdate)

{

foreach (var subscriber in \_subscribers)

{

subscriber.Update(newsUpdate);

}

}

}

**Step 3: Define the Subscriber Interface (INewsSubscriber)**

The INewsSubscriber interface defines the contract for subscribers. It includes a method Update that will be called when the publisher notifies about news updates.

public interface INewsSubscriber

{

void Update(string newsUpdate);

}

**Step 4: Implement Concrete Subscribers (EmailSubscriber and SmsSubscriber)**

The EmailSubscriber and SmsSubscriber classes implement the INewsSubscriber interface. They represent different types of subscribers who receive news updates through emails and SMS, respectively.

public class EmailSubscriber : INewsSubscriber

{

**// ...**

public void Update(string newsUpdate)

{

Console.WriteLine($"Sending email to {\_email}: {newsUpdate}");

}

}

public class SmsSubscriber : INewsSubscriber

{

**// ...**

public void Update(string newsUpdate)

{

Console.WriteLine($"Sending SMS to {\_phoneNumber}: {newsUpdate}");

}

}

**Step 5: Client Code (Main Method)**

In the Main method, you create instances of the publisher (NewsAgency) and subscribers (EmailSubscriber and SmsSubscriber). You then add subscribers to the publisher's list and notify them about a news update.

static void Main(string[] args)

{

NewsAgency newsAgency = new NewsAgency();

INewsSubscriber emailSubscriber = new EmailSubscriber("user@example.com");

INewsSubscriber smsSubscriber = new SmsSubscriber("+1234567890");

newsAgency.AddSubscriber(emailSubscriber);

newsAgency.AddSubscriber(smsSubscriber);

newsAgency.NotifySubscribers("Breaking News: Important announcement!");

}

**Explanation:**

**Observer Pattern**: The Observer pattern defines a one-to-many relationship between objects, where changes in one object (the subject) are automatically reflected in other objects (observers).

**Step 1**: The INewsPublisher interface represents the subject (publisher), defining methods to manage subscribers and notify them.

**Step 2**: The NewsAgency class (concrete publisher) maintains a list of subscribers and sends them news updates.

**Step 3**: The INewsSubscriber interface defines the observer, providing a method for subscribers to receive updates.

**Step 4**: The EmailSubscriber and SmsSubscriber classes (concrete observers) implement the subscriber interface to receive and handle updates.

**Step 5**: In the Main method, you create instances of the publisher and subscribers, add subscribers to the publisher, and notify them about a news update. This demonstrates the Observer pattern, where subscribers (observers) are automatically notified when the publisher's state changes.

By using the Observer pattern, your code establishes a communication mechanism where objects (subscribers) are notified of changes in another object's (publisher's) state, allowing for loose coupling and dynamic relationships between components.