Lab Exercises: Delegates and Events

What is a Delegate?

A delegate is a type that represents references to methods with a particular parameter list and return type. When you instantiate a delegate, you can associate its instance with any method with a compatible signature and return type. You can invoke the method through the delegate instance.

Use Cases for Delegates

1. **Callback Methods**: Useful in scenarios where methods need to be called back at a later time, such as asynchronous programming or event-driven programming.
2. **Encapsulating Method References**: Allow passing methods as parameters.
3. **LINQ and Functional Programming**: Used extensively in LINQ operations and other functional programming scenarios.
4. **Event Handling**: Delegates are the foundation for defining events and event handlers in .NET.

Lab Exercise: Implementing Delegates and Events in a Real-World Scenario

Scenario

You are tasked with building a simple stock trading application. This application needs to notify users when the price of a stock changes. You'll implement this using delegates and events.

Step-by-Step Lab Exercise

1. **Create the Project**
   - Open Visual Studio and create a new Console Application project named `StockTradingApp`.
2. **Define a Delegate**
   - Create a delegate named `PriceChangedHandler` that takes two parameters: the stock symbol and the new price.

```
public delegate void PriceChangedHandler(string symbol, decimal
newPrice);
```

3. **Create a Stock Class**
   - Create a class named `Stock` with the following:
     - Properties: `Symbol`, `Price`.
     - Event: `PriceChanged`, using the delegate `PriceChangedHandler`.
     - Method: `UpdatePrice(decimal newPrice)` to update the stock price and trigger the event if the price changes.

```
public class Stock
{
    public string Symbol { get; }
```

```
        private decimal _price;
        public decimal Price
        {
            get => _price;
            private set
            {
                if (_price != value)
                {
                    _price = value;
                    OnPriceChanged(value);
                }
            }
        }

        public event PriceChangedHandler PriceChanged;

        public Stock(string symbol, decimal initialPrice)
        {
            Symbol = symbol;
            _price = initialPrice;
        }

        protected virtual void OnPriceChanged(decimal newPrice)
        {
            PriceChanged?.Invoke(Symbol, newPrice);
        }

        public void UpdatePrice(decimal newPrice)
        {
            Price = newPrice;
        }
    }
```

4. **Create the Stock Market Class**
    - Create a class named `StockMarket` that manages multiple stocks.
    - Add a method to add stocks and a method to update stock prices.

```
    public class StockMarket
    {
        private readonly List<Stock> _stocks = new List<Stock>();

        public void AddStock(Stock stock)
        {
            _stocks.Add(stock);
        }

        public void UpdateStockPrice(string symbol, decimal newPrice)
        {
            var stock = _stocks.FirstOrDefault(s => s.Symbol == symbol);
            if (stock != null)
            {
                stock.UpdatePrice(newPrice);
            }
        }
    }
```

5. **Create a Subscriber Class**
    - Create a class named `StockSubscriber` that subscribes to the `PriceChanged` event of a `Stock` object and prints the new price to the console.

```
public class StockSubscriber
{
    public void Subscribe(Stock stock)
    {
        stock.PriceChanged += OnPriceChanged;
    }

    private void OnPriceChanged(string symbol, decimal newPrice)
    {
        Console.WriteLine($"The price of {symbol} has changed to 
{newPrice}");
    }
}
```

6. **Test the Application**
   - In the `Main` method of your `Program` class, create an instance of `StockMarket`, add some stocks, and subscribe to the `PriceChanged` event using `StockSubscriber`. Then, update the stock prices to see the event in action.

```
class Program
{
    static void Main(string[] args)
    {
        StockMarket market = new StockMarket();
        StockSubscriber subscriber = new StockSubscriber();

        Stock apple = new Stock("AAPL", 150.00m);
        Stock google = new Stock("GOOGL", 2800.00m);

        subscriber.Subscribe(apple);
        subscriber.Subscribe(google);

        market.AddStock(apple);
        market.AddStock(google);

        market.UpdateStockPrice("AAPL", 155.00m);
        market.UpdateStockPrice("GOOGL", 2900.00m);
    }
}
```

Explanation of Steps

1. **Creating the Project**: This step involves setting up the environment for your console application.
2. **Defining a Delegate**: The delegate `PriceChangedHandler` is defined to represent methods that handle price changes.
3. **Creating the Stock Class**: The `Stock` class encapsulates stock data and triggers the `PriceChanged` event whenever the price changes.
4. **Creating the Stock Market Class**: This class manages a collection of stocks and allows for updating their prices.
5. **Creating a Subscriber Class**: This class demonstrates how to subscribe to events and handle them.
6. **Testing the Application**: This step ties everything together and tests the functionality of the delegates and events implementation.

Conclusion

This lab exercise demonstrates how to use delegates and events to implement a real-world scenario. You have learned how to define a delegate, create and trigger events, and handle those events in subscriber classes.