LOUISVILLE
.NET MEETUP

Louisville-DotNet

@LouDotNet

Upcoming Events

Upcoming Meetups

Date	Session
April 17	TBD
May 15	Build agents and extend copilot experiences with Copilot Studio
June 26	TBD

Continue the Conversation

The Dark Side of Design Patterns: What They Don’t Tell You

Who is Chad Green?

✉ chadgreen@chadgreen.com

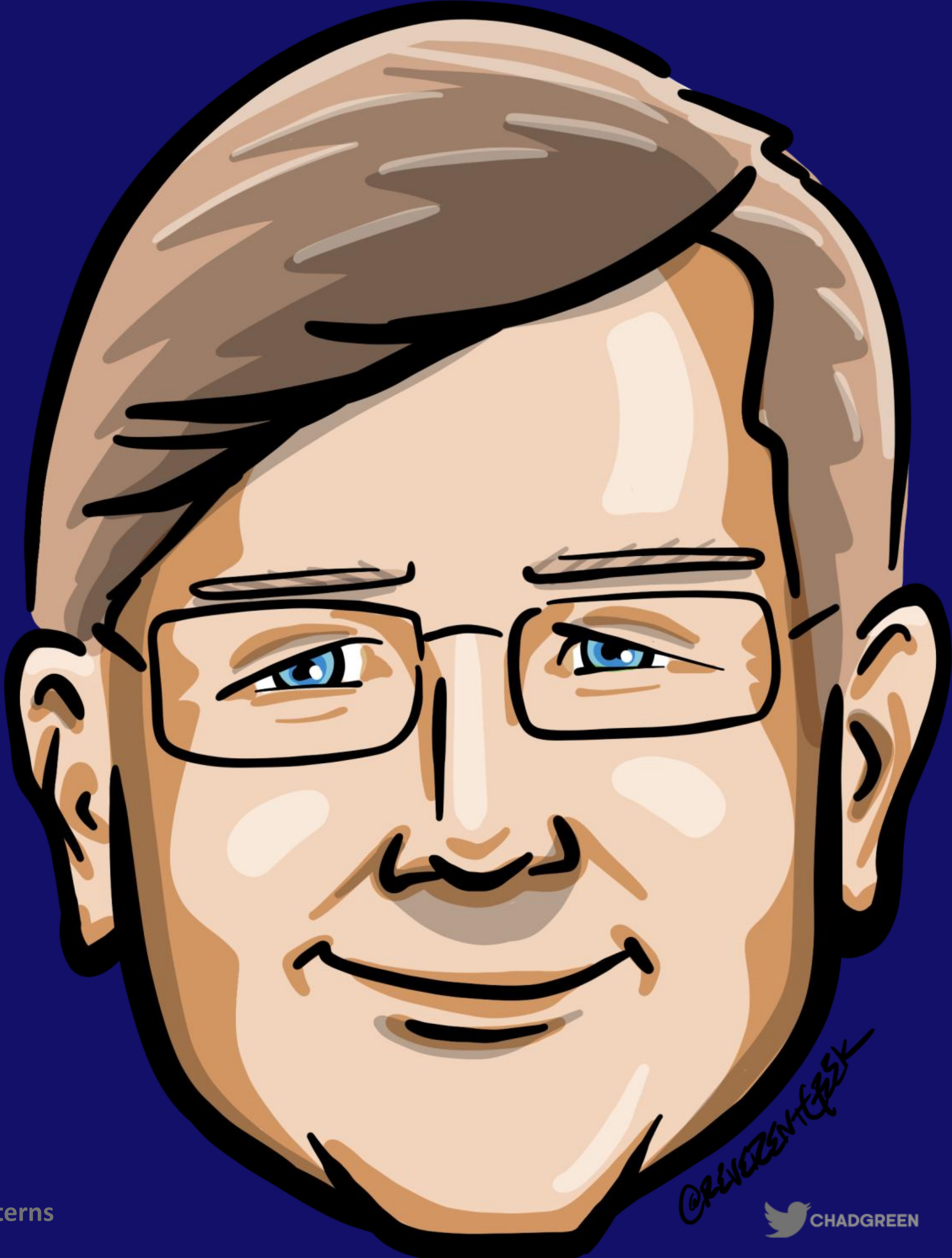
TaleLearnCode

ChadGreen.com

ChadGreen & TaleLearnCode

in ChadwickEGreen

MVP Microsoft® Most Valuable Professional



The Power of Design Patterns

Reevaluating Software Design Patterns

Significance of Design Patterns

Code Reusability

Significance of Design Patterns

Code Reusability

Scalability and Maintainability

Significance of Design Patterns

Code Reusability

Scalability and Maintainability

Common Vocabulary

Significance of Design Patterns

- Code Reusability
- Scalability and Maintainability
- Common Vocabulary
- **Best Practices**

Significance of Design Patterns

Code Reusability

Scalability and Maintainability

Common Vocabulary

Best Practices

Abstraction and Flexibility

Significance of Design Patterns

Code Reusability

Scalability and Maintainability

Common Vocabulary

Best Practices

Abstraction and Flexibility

Ease of Maintenance

Significance of Design Patterns

- Code Reusability
- Scalability and Maintainability
- Common Vocabulary
- Best Practices
- Abstraction and Flexibility
- Ease of Maintenance
- **Learning and Onboarding**

Significance of Design Patterns

Code Reusability

Scalability and Maintainability

Common Vocabulary

Best Practices

Abstraction and Flexibility

Ease of Maintenance

Learning and Onboarding

Documentation

Significance of Design Patterns

Code Reusability

Scalability and Maintainability

Common Vocabulary

Best Practices

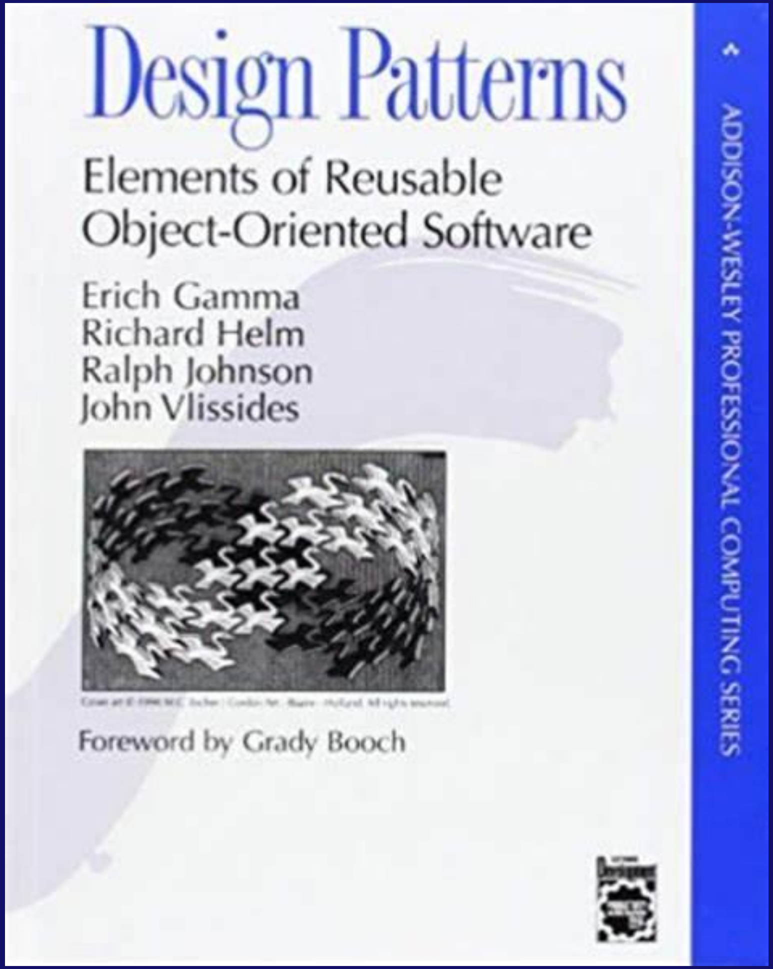
Abstraction and Flexibility

Ease of Maintenance

Learning and Onboarding

Documentation

Gang of Four



Design Patterns

Elements of Reusable Object-Oriented Software

Erich Gamma
Richard Helm
Ralph Johnson
John Vlissides

Foreword by Grady Booch

ADDISON-WESLEY PROFESSIONAL COMPUTING SERIES

Main Types of Design Patterns

Creation

- Interpreter
- Template Method
- Chain of Responsibility
- Command
- Iterator
- Mediator
- Memento
- Observer
- State
- Strategy
- Visitor

Main Types of Design Patterns

Creation

Structural

- Factory Method
- Abstract Factory
- Builder

- Prototype
- Singleton

Main Types of Design Patterns

Creation

Structural

Behavioral

- Adapter
- Bridge
- Composite
- Decorator

- Façade
- Flyweight
- Proxy

Main Types of Design Patterns

Creation

Structural

Behavioral

Architectural

- Model-View-Controller (MVC)
- Layered Architecture
- Microservices
- Event-Driven Architecture
- Service-Oriented Architecture

Not All Patterns Are Created Equal

Reevaluating Software Design Patterns

Not all patterns are created equal

- Should be applied judiciously

Not all patterns are created equal

- Should be applied judiciously
- **Appropriateness influenced by nature of software being developed**

Not all patterns are created equal

- Should be applied judiciously
- Appropriateness influenced by nature of software being developed
- **Essential to carefully evaluate trade-offs**

Not all patterns are created equal

- Should be applied judiciously
- Appropriateness influenced by nature of software being developed
- Essential to carefully evaluate trade-offs

The Problematic Patterns

Reevaluating Software Design Patterns

Not talking about anti-patterns

- God Object
- Spaghetti Code
- Copy-Paste Programming
- Magic Numbers
- Hard Coding
- Lava Flow
- Circular Dependency
- Premature Optimization

The Problematic Patterns

- Singleton
- Observer
- Factory

Singleton Pattern

Reevaluating Software Design Patterns

Singleton Pattern

Single Instance

Singleton Pattern

Single Instance

Global Access

Singleton Pattern

Single Instance

Global Access

Lazy Initialization

Singleton Pattern

Single Instance

Global Access

Lazy Initialization

Private Constructor

Singleton Pattern

Single Instance

Global Access

Lazy Initialization

Private Constructor

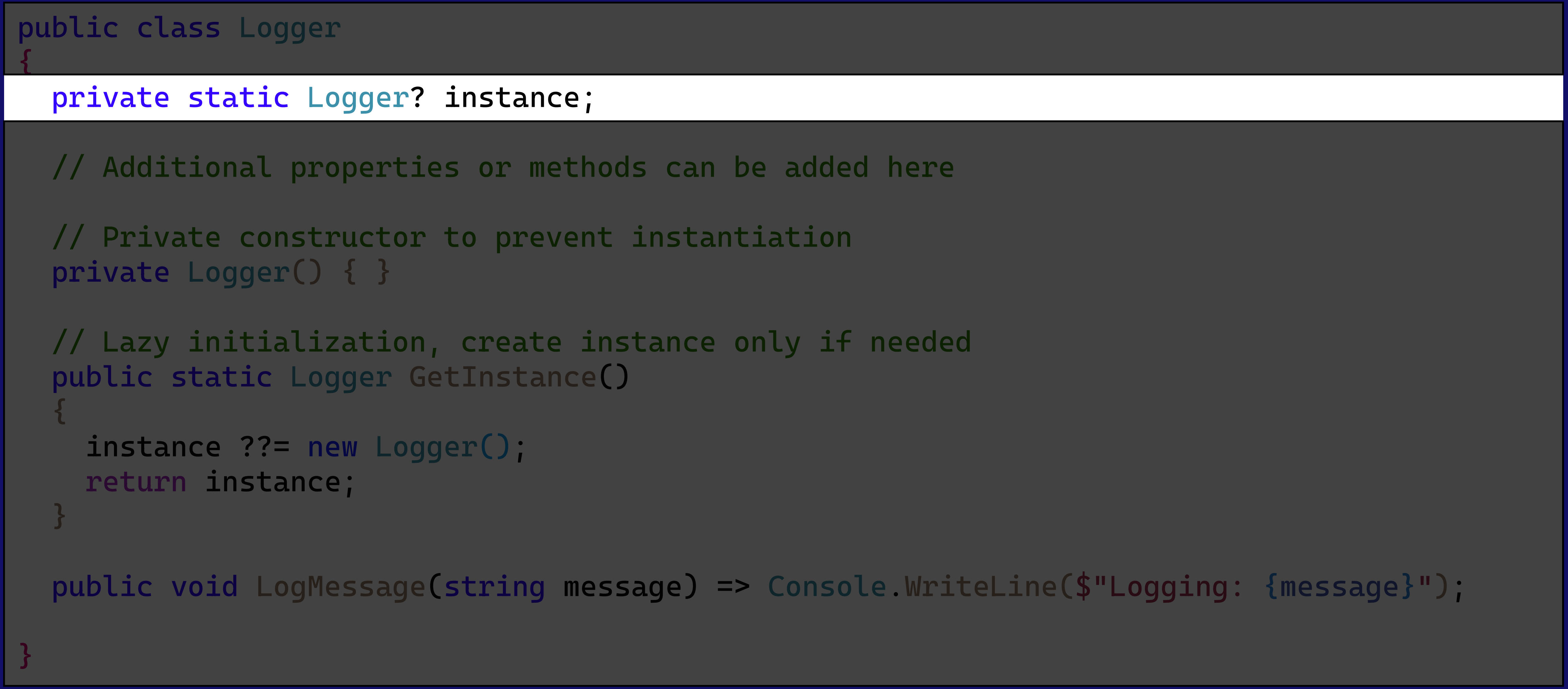
Static Instance Method/Property

Demo: Singleton Pattern

Singleton Class

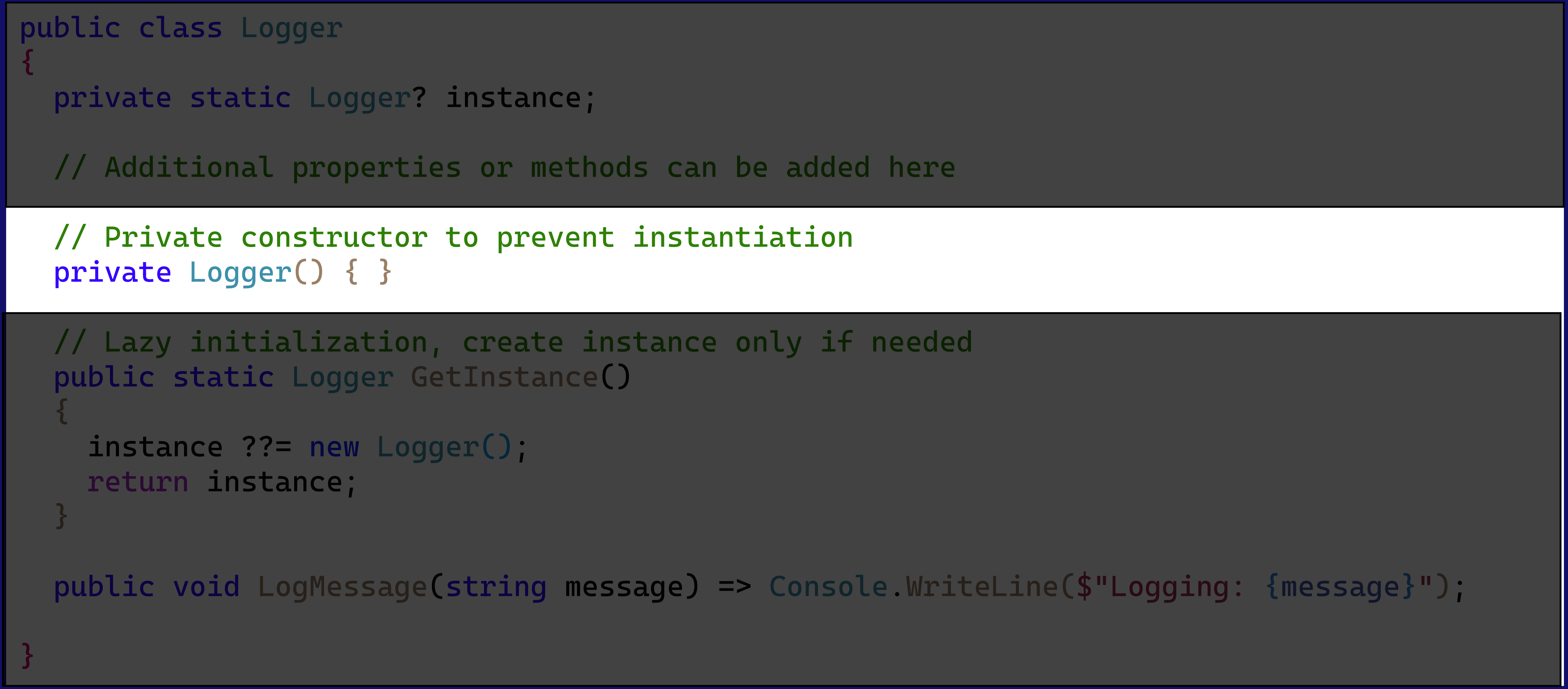
```csharp
public class Logger
{
  private static Logger? instance;

  // Additional properties or methods can be added here

  // Private constructor to prevent instantiation
  private Logger() { }

  // Lazy initialization, create instance only if needed
  public static Logger GetInstance()
  {
    instance ??= new Logger();
    return instance;
  }

  public void LogMessage(string message) => Console.WriteLine($"Logging: {message}");

}
```

Singleton Class



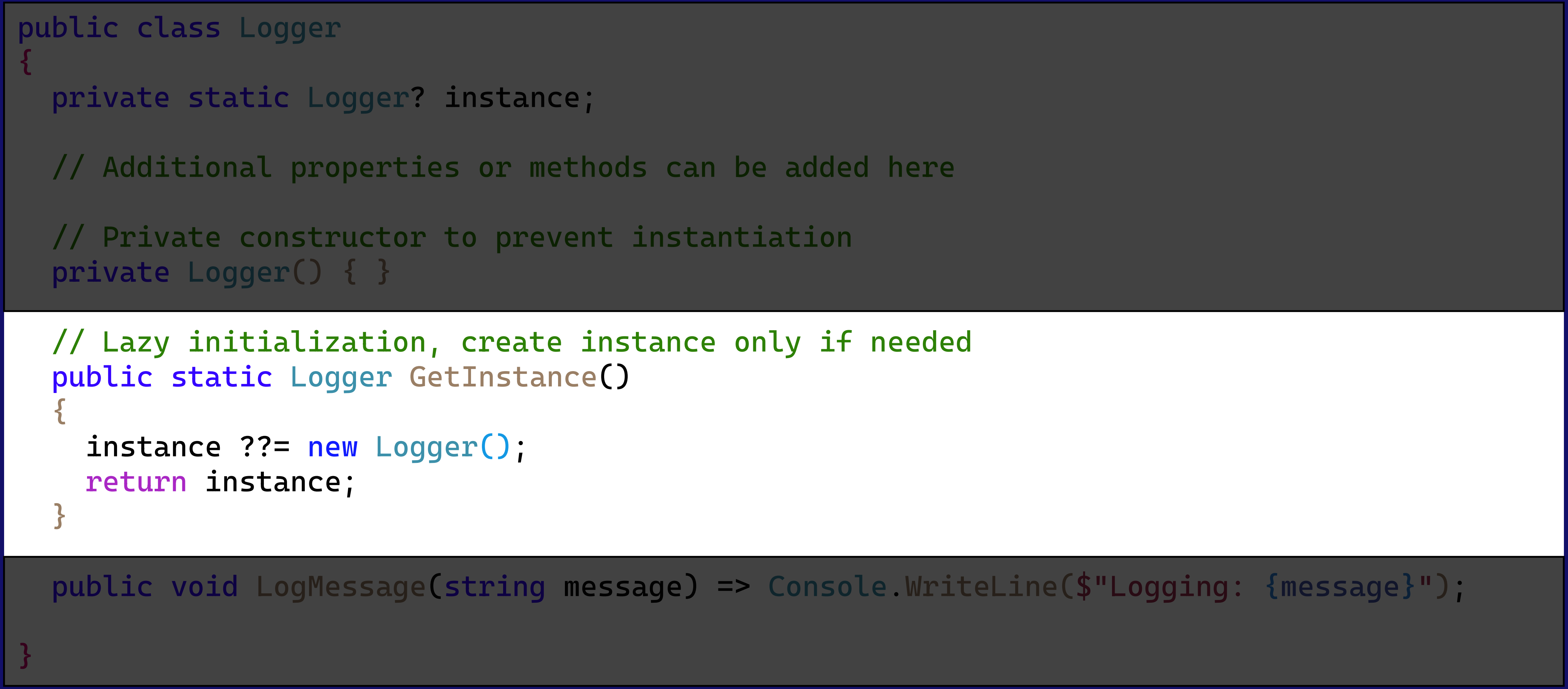
```csharp
public class Logger
{
  private static Logger? instance;

  // Additional properties or methods can be added here

  // Private constructor to prevent instantiation
  private Logger() { }

  // Lazy initialization, create instance only if needed
  public static Logger GetInstance()
  {
    instance ??= new Logger();
    return instance;
  }

  public void LogMessage(string message) => Console.WriteLine($"Logging: {message}");

}
```

Singleton Class



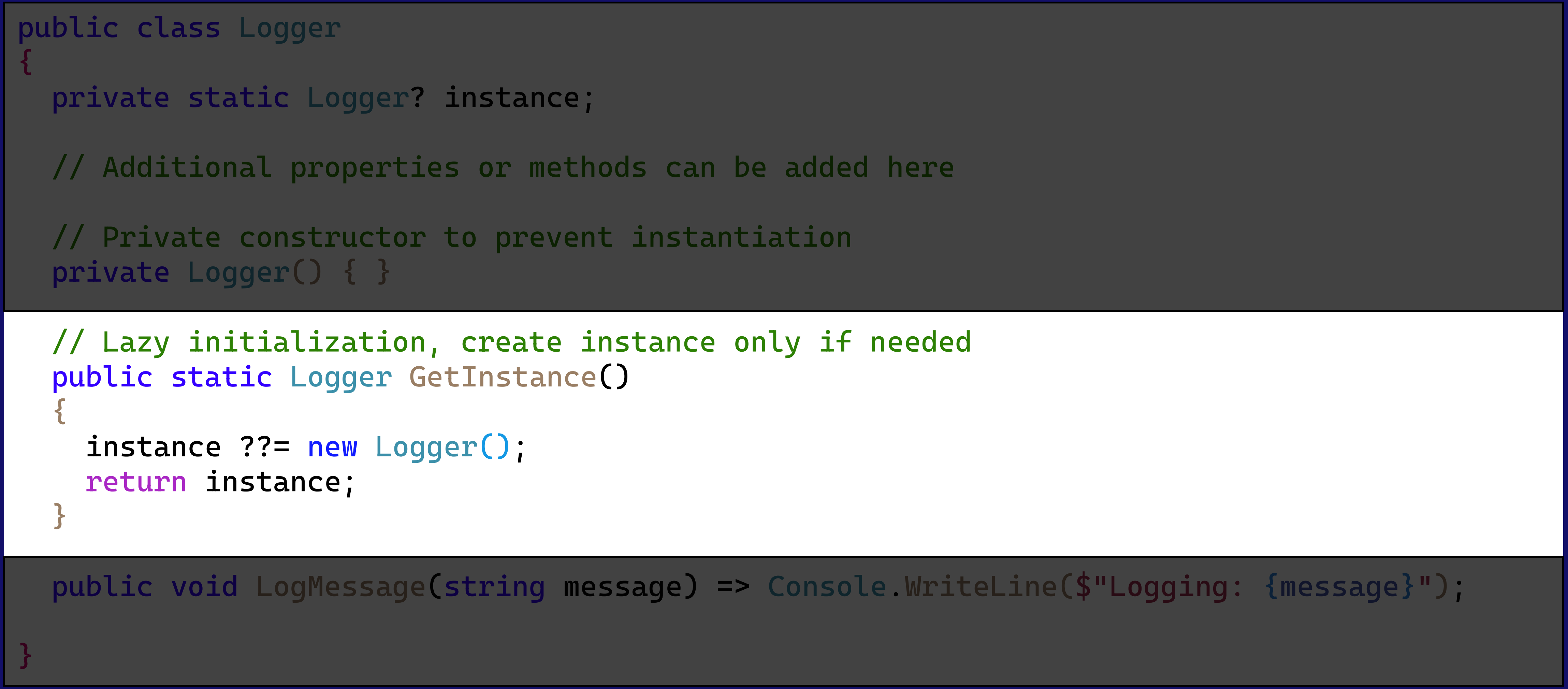
```
public class Logger
{
  private static Logger? instance;

  // Additional properties or methods can be added here

  // Private constructor to prevent instantiation
  private Logger() { }

  // Lazy initialization, create instance only if needed
  public static Logger GetInstance()
  {
    instance ??= new Logger();
    return instance;
  }

  public void LogMessage(string message) => Console.WriteLine($"Logging: {message}");

}
```

Singleton Class



```
public class Logger
{
  private static Logger? instance;

  // Additional properties or methods can be added here

  // Private constructor to prevent instantiation
  private Logger() { }

  // Lazy initialization, create instance only if needed
  public static Logger GetInstance()
  {
    instance ??= new Logger();
    return instance;
  }

  public void LogMessage(string message) => Console.WriteLine($"Logging: {message}");

}
```

Singleton Class



```
public class Logger
{
  private static Logger? instance;

  // Additional properties or methods can be added here

  // Private constructor to prevent instantiation
  private Logger() { }

  // Lazy initialization, create instance only if needed
  public static Logger GetInstance()
  {
    instance ??= new Logger();
    return instance;
  }

  public void LogMessage(string message) => Console.WriteLine($"Logging: {message}");

}
```

Main Object

```
// Using the Singleton Logger
Logger logger = Logger.GetInstance();
logger.LogMessage("Application started");

// Using the Singleton Logger within a service
UserService userService = new();
userService.PerformUserAction("JohnDoe", "Login");

// Ensure that the same logger instance is used throughout the application
Logger anotherLogger = Logger.GetInstance();
Console.WriteLine($“Same instance? {ReferenceEquals(logger, anotherLogger)}");
```

Main Object

```
// Using the Singleton Logger
Logger logger = Logger.GetInstance();
logger.LogMessage("Application started");
```

```
// Using the Singleton Logger within a service
UserService userService = new();
userService.PerformUserAction("JohnDoe", "Login");

// Ensure that the same logger instance is used throughout the application
Logger anotherLogger = Logger.GetInstance();
Console.WriteLine($“Same instance? {ReferenceEquals(logger, anotherLogger)}");
```

Main Object

```
// Using the Singleton Logger
Logger logger = Logger.GetInstance();
logger.LogMessage("Application started");

// Using the Singleton Logger within a service
UserService userService = new();
userService.PerformUserAction("JohnDoe", "Login");

// Ensure that the same logger instance is used throughout the application
Logger anotherLogger = Logger.GetInstance();
Console.WriteLine($“Same instance? {ReferenceEquals(logger, anotherLogger)}");
```

Another Object

```
public class UserService
{

  private readonly Logger logger;

  public UserService()
  {
    logger = Logger.GetInstance();
  }

  public void PerformUserAction(string userName, string action)
  {
    // Some business logic
    logger.LogMessage($"User '{userName}' performed action: {action}");
  }
}
```

Main Object

```
// Using the Singleton Logger
Logger logger = Logger.GetInstance();
logger.LogMessage("Application started");

// Using the Singleton Logger within a service
UserService userService = new();
userService.PerformUserAction("JohnDoe", "Login");

// Ensure that the same logger instance is used throughout the application
Logger anotherLogger = Logger.GetInstance();
Console.WriteLine($“Same instance? {ReferenceEquals(logger, anotherLogger)}");
```

Singleton Pattern: The Good

Centralized Logging

Singleton Pattern: The Good

Centralized Logging

Global Access to Logger

Singleton Pattern: The Good

Centralized Logging

Global Access to Logger

Lazy Initialization

Singleton Pattern: The Good

Centralized Logging

Global Access to Logger

Lazy Initialization

Instance Reusability

Singleton Pattern: The Good

Centralized Logging

Global Access to Logger

Lazy Initialization

Instance Reusability

Straightforward Usage

Singleton Pattern: The Good

Centralized Logging

Global Access to Logger

Lazy Initialization

Instance Reusability

Straightforward Usage

Simple Initialization

Singleton Pattern: The Good

- Centralized Logging
- Global Access to Logger
- Lazy Initialization
- Instance Reusability
- Straightforward Usage
- Simple Initialization

Singleton Pattern: The Bad

Global State

Singleton Pattern: The Bad

Global State

Tight Coupling

Singleton Pattern: The Bad

Global State

Tight Coupling

Testing Challenges

Singleton Pattern: The Bad

- Global State
- Tight Coupling
- Testing Challenges
- Hidden Dependencies

Singleton Pattern: The Bad

- Global State
- Tight Coupling
- Testing Challenges
- Hidden Dependencies
- Inflexible Initialization

Singleton Pattern: The Bad

- Global State
- Tight Coupling
- Testing Challenges
- Hidden Dependencies
- Inflexible Initialization
- Thread Safety Issues

Singleton Pattern: The Bad

Global State

Tight Coupling

Testing Challenges

Hidden Dependencies

Inflexible Initialization

Thread Safety Issues

- Race Conditions

Singleton Pattern: The Bad

Global State

Tight Coupling

Testing Challenges

Hidden Dependencies

Inflexible Initialization

Thread Safety Issues

- Race Conditions
- **Double-Checked Locking**

Singleton Pattern: The Bad

Global State

Tight Coupling

Testing Challenges

Hidden Dependencies

Inflexible Initialization

Thread Safety Issues

- Race Conditions
- Double-Checked Locking
- **Synchronization Overhead**

Singleton Pattern: The Bad

Global State

Tight Coupling

Testing Challenges

Hidden Dependencies

Inflexible Initialization

Thread Safety Issues

- Race Conditions
- Double-Checked Locking
- Synchronization Overhead
- **Deadlocks**

Singleton Pattern: The Bad

Global State

Tight Coupling

Testing Challenges

Hidden Dependencies

Inflexible Initialization

Thread Safety Issues

- Race Conditions
- Double-Checked Locking
- Synchronization Overhead
- Deadlocks
- **Resource Management**

Singleton Pattern: The Bad

Global State

Tight Coupling

Testing Challenges

Hidden Dependencies

Inflexible Initialization

Non-Thread Safe Init

Potential for Misuse

Singleton Pattern: The Bad

- Global State
- Tight Coupling
- Testing Challenges
- Hidden Dependencies
- Inflexible Initialization
- Non-Thread Safe Init
- Potential for Misuse

Alternatives/Modifications

- Dependency Injection

Alternatives/Modifications

- Dependency Injection
- **Factory Method Pattern**

Alternatives/Modifications

- Dependency Injection
- Factory Method Pattern
- **Service Locator Pattern**

Alternatives/Modifications

- Dependency Injection
- Factory Method Pattern
- Service Locator Pattern
- **Inversion of Control (IoC) Containers**

Alternatives/Modifications

- Dependency Injection
- Factory Method Pattern
- Service Locator Pattern
- Inversion of Control (IoC) Containers
- **Prototype Pattern**

Alternatives/Modifications

- Dependency Injection
- Factory Method Pattern
- Service Locator Pattern
- Inversion of Control (IoC) Containers
- Prototype Pattern
- **Thread-Safe Singleton Initialization**

Alternatives/Modifications

- Dependency Injection
- Factory Method Pattern
- Service Locator Pattern
- Inversion of Control (IoC) Containers
- Prototype Pattern
- Thread-Safe Singleton Initialization
- **Enum Singleton**

Alternatives/Modifications

- Dependency Injection
- Factory Method Pattern
- Service Locator Pattern
- Inversion of Control (IoC) Containers
- Prototype Pattern
- Thread-Safe Singleton Initialization
- Enum Singleton
- **Immutable Objects**

Alternatives/Modifications

- Dependency Injection
- Factory Method Pattern
- Service Locator Pattern
- Inversion of Control (IoC) Containers
- Prototype Pattern
- Thread-Safe Singleton Initialization
- Enum Singleton
- Immutable Objects

Alternatives/Modifications

- Dependency Injection
- Factory Method Pattern
- Service Locator Pattern
- Inversion of Control (IoC) Containers
- Prototype Pattern
- Thread-Safe Singleton Initialization
- Enum Singleton
- Immutable Objects

Observer Pattern

Reevaluating Software Design Patterns

Observer Pattern

Key Components

- Subject

Observer Pattern

Key Components

- Subject
- Observer

Observer Pattern

Key Components

- Subject
- Observer
- Concrete Subject

Observer Pattern

Key Components

- Subject
- Observer
- Concrete Subject
- Concrete Observer

Observer Pattern

Key Components

- Subject
- Observer
- Concrete Subject
- Concrete Observer

Workflow

Observer Pattern

Key Components

- Subject
- Observer
- Concrete Subject
- Concrete Observer

Workflow

- Registration

Observer Pattern

Key Components

- Subject
- Observer
- Concrete Subject
- Concrete Observer

Workflow

- Registration
- Notification

Observer Pattern

Key Components

- Subject
- Observer
- Concrete Subject
- Concrete Observer

Workflow

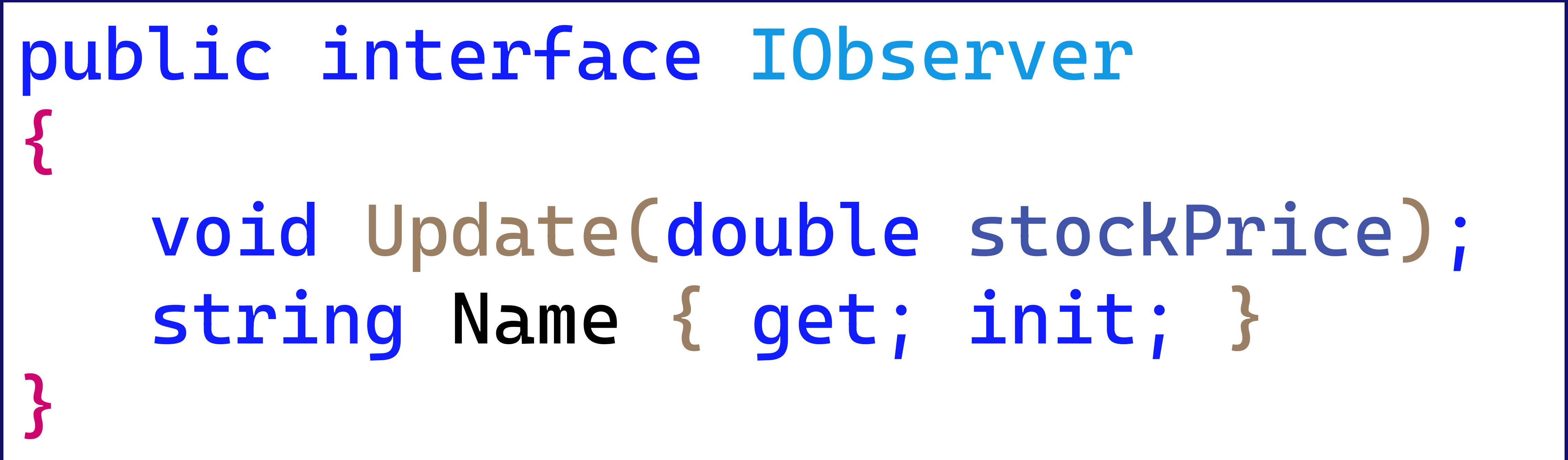
- Registration
- Notification
- Update

Demo: Observer Pattern

Subject

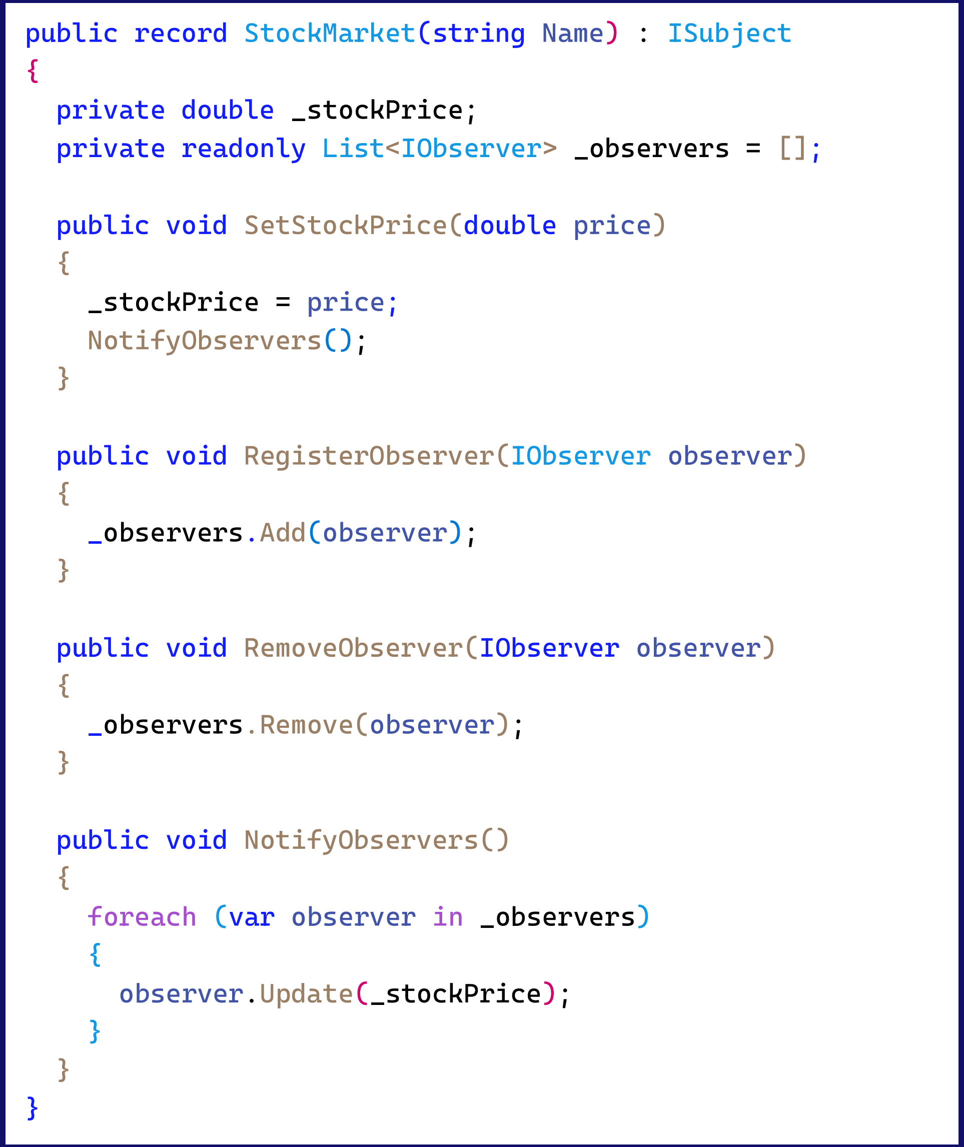
```
public interface ISubject
{
  void RegisterObserver(IObserver observer);
  void RemoveObserver(IObserver observer);
  void NotifyObservers();
  string Name { get; init; }
}
```

Observer



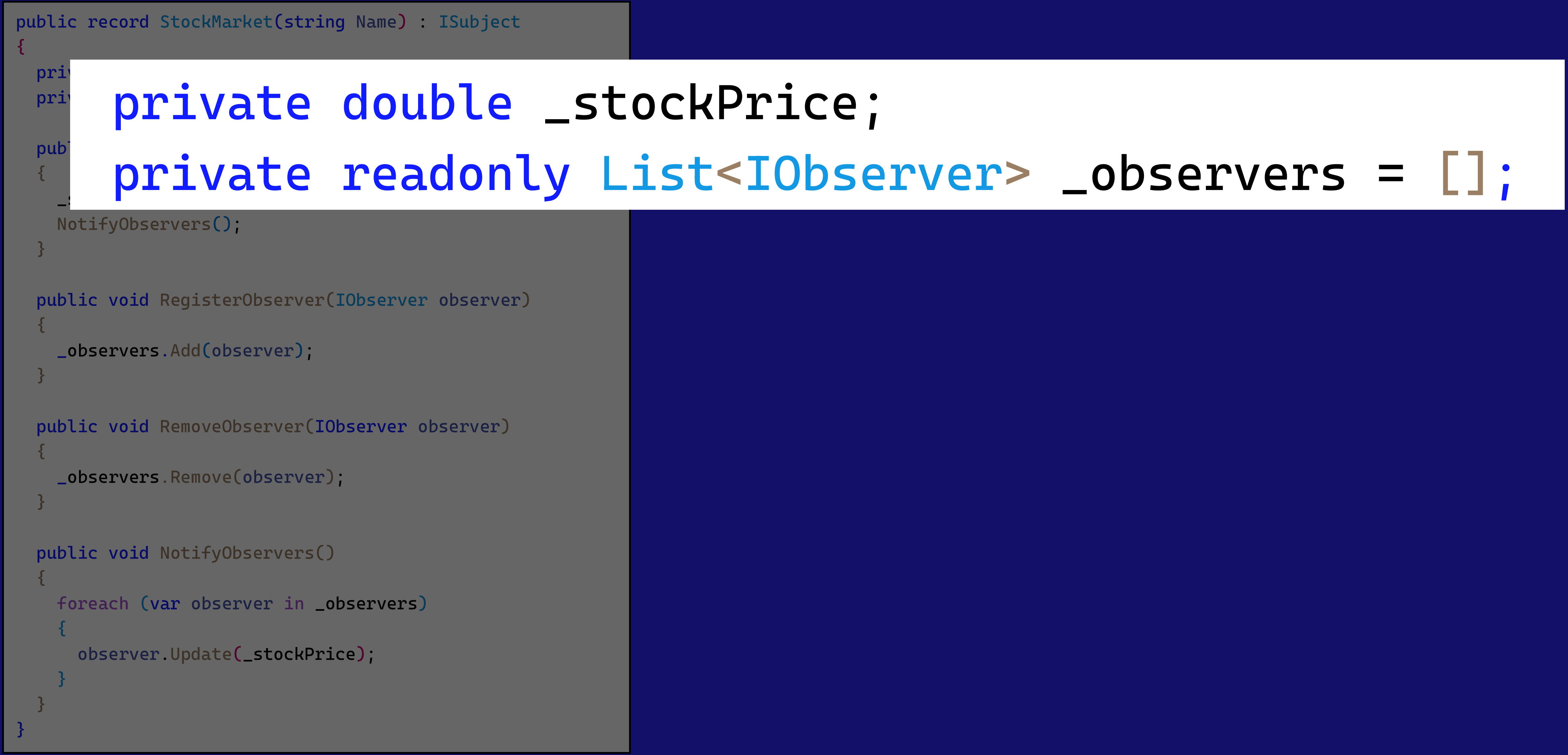
```
public interface IObserver
{
    void Update(double stockPrice);
    string Name { get; init; }
}
```

Concrete Subject



```
public record StockMarket(string Name) : ISubject
{
  private double _stockPrice;
  private readonly List<IObserver> _observers = [];

  public void SetStockPrice(double price)
  {
    _stockPrice = price;
    NotifyObservers();
  }

  public void RegisterObserver(IObserver observer)
  {
    _observers.Add(observer);
  }

  public void RemoveObserver(IObserver observer)
  {
    _observers.Remove(observer);
  }

  public void NotifyObservers()
  {
    foreach (var observer in _observers)
    {
      observer.Update(_stockPrice);
    }
  }
}
```

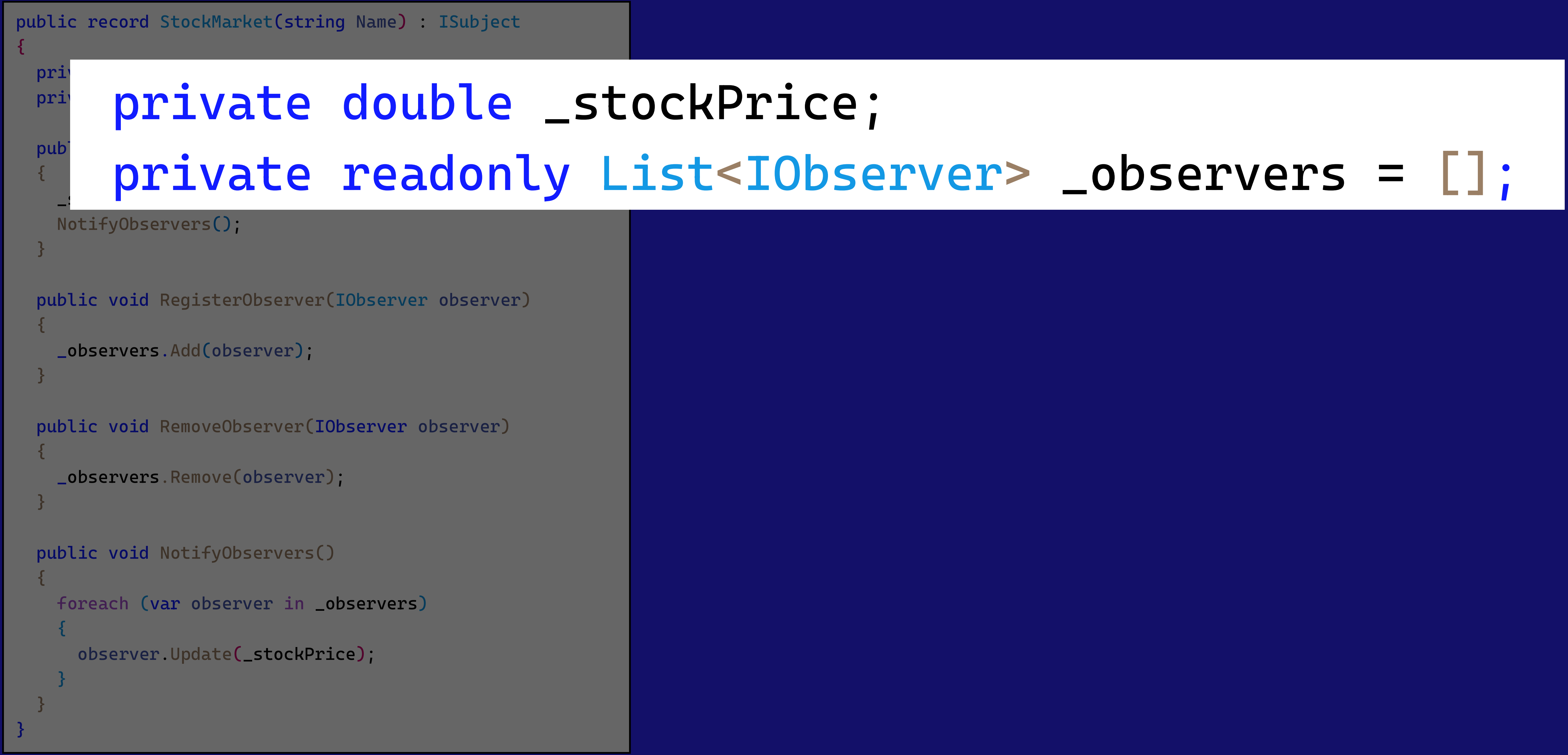
Concrete Subject



```
public record StockMarket(string Name) : ISubject
{
  pri
  pri

  pub
  {
    _
    NotifyObservers();
  }

  public void RegisterObserver(IObserver observer)
  {
    _observers.Add(observer);
  }

  public void RemoveObserver(IObserver observer)
  {
    _observers.Remove(observer);
  }

  public void NotifyObservers()
  {
    foreach (var observer in _observers)
    {
      observer.Update(_stockPrice);
    }
  }
}
```

```
private double _stockPrice;
private readonly List<IObserver> _observers = [];
```

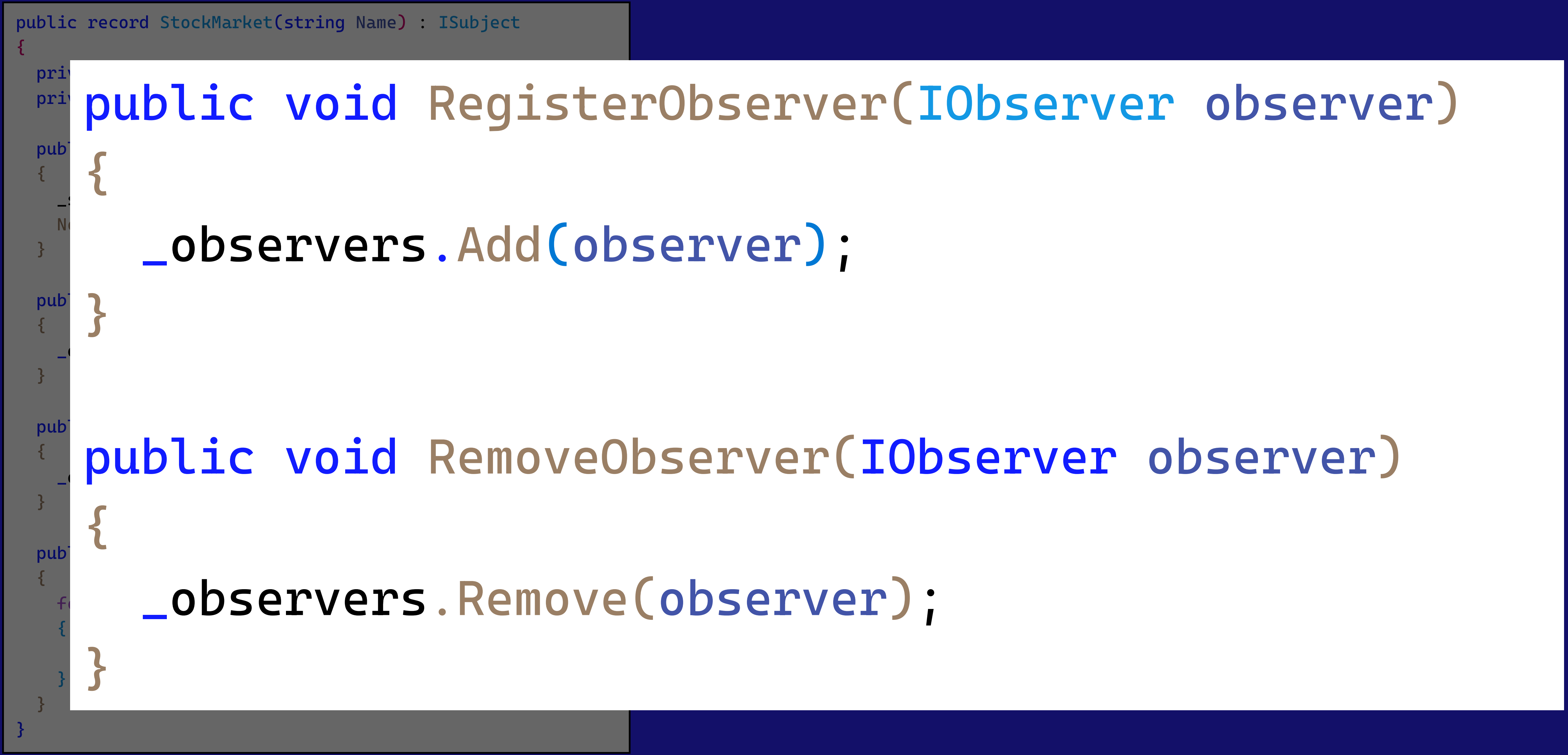
Concrete Subject



```
public record StockMarket(string Name) : ISubject
{
  pri
  pri

  pub
  {
    _s
    NotifyObservers();
  }

  public void RegisterObserver(IObserver observer)
  {
    _observers.Add(observer);
  }

  public void RemoveObserver(IObserver observer)
  {
    _observers.Remove(observer);
  }

  public void NotifyObservers()
  {
    foreach (var observer in _observers)
    {
      observer.Update(_stockPrice);
    }
  }
}
```

```
private double _stockPrice;
private readonly List<IObserver> _observers = [];
```

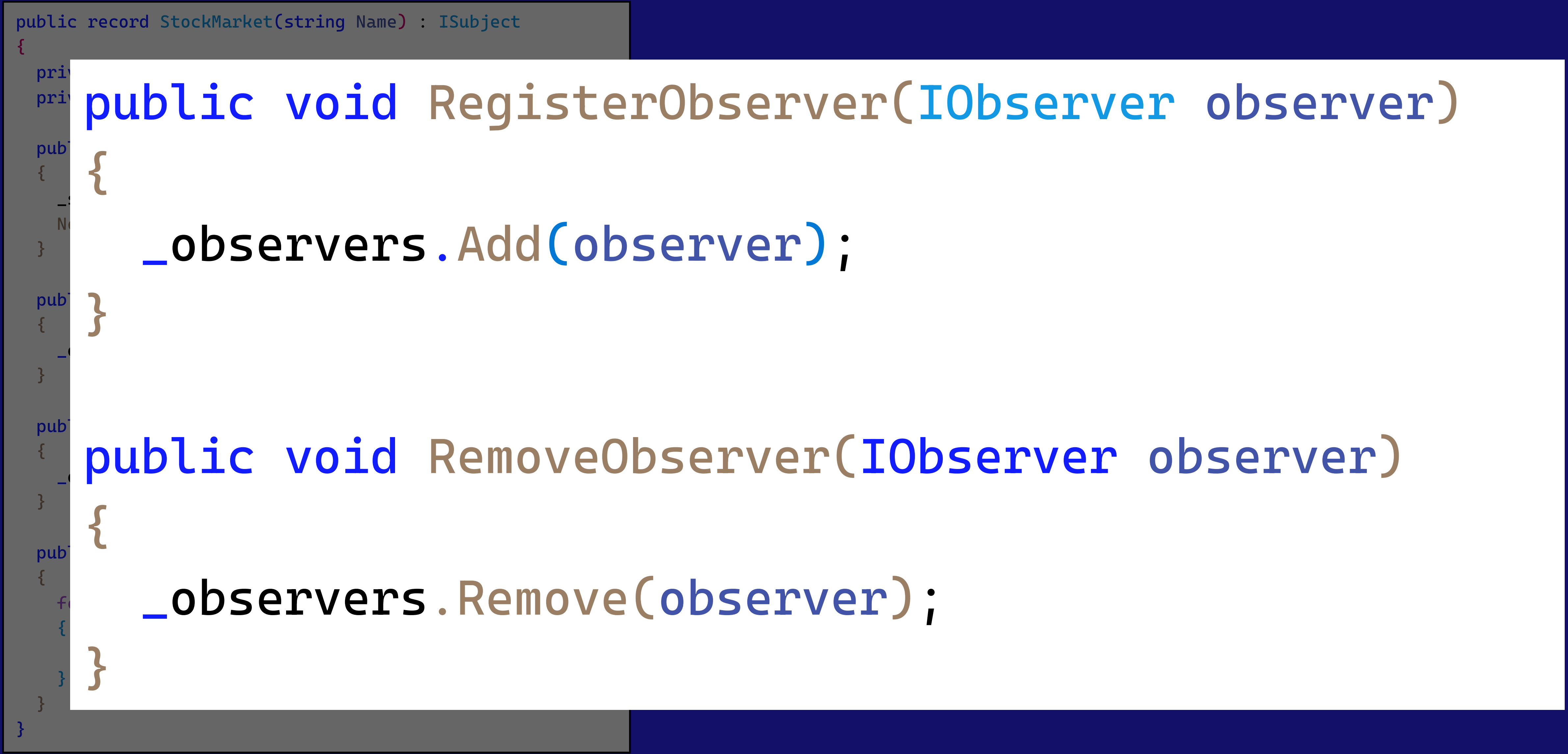
Concrete Subject



```
public record StockMarket(string Name) : ISubject
{
    pri
    pri

    pub
    {
        _
        N
    }

    pub
    {
        _
    }

    pub
    {
        _
    }

    pub
    {
        f
        {

        }
    }
}
```

```
public void RegisterObserver(IObserver observer)
{
   _observers.Add(observer);
}


public void RemoveObserver(IObserver observer)
{
   _observers.Remove(observer);
}
```

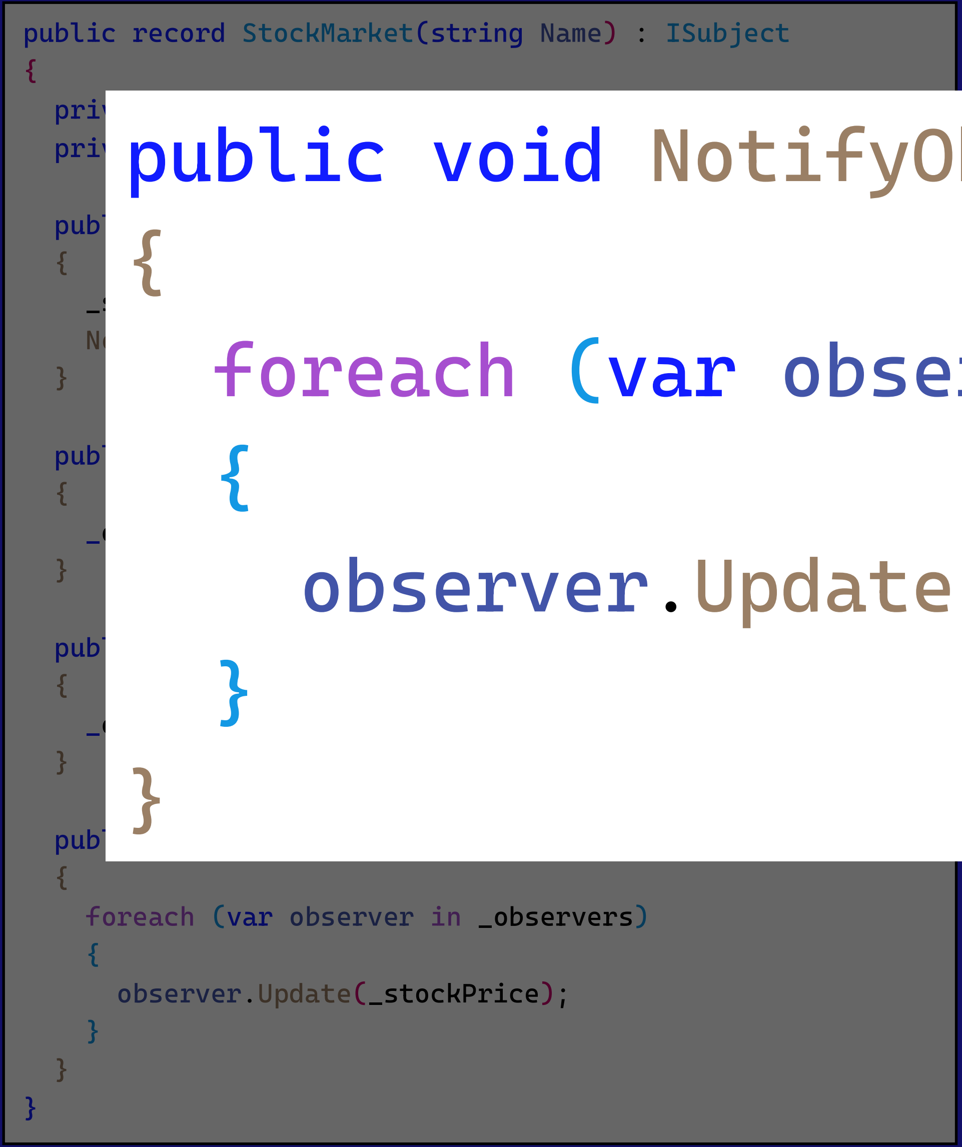
Concrete Subject



```
public record StockMarket(string Name) : ISubject
{
  pri
  pri

  pub
  {
    _
    N
  }

  pub
  {
    _
  }

  pub
  {
    _
  }

  pub
  {
    f
    {

    }
  }
}
```

```
public void RegisterObserver(IObserver observer)
{
  _observers.Add(observer);
}

public void RemoveObserver(IObserver observer)
{
  _observers.Remove(observer);
}
```

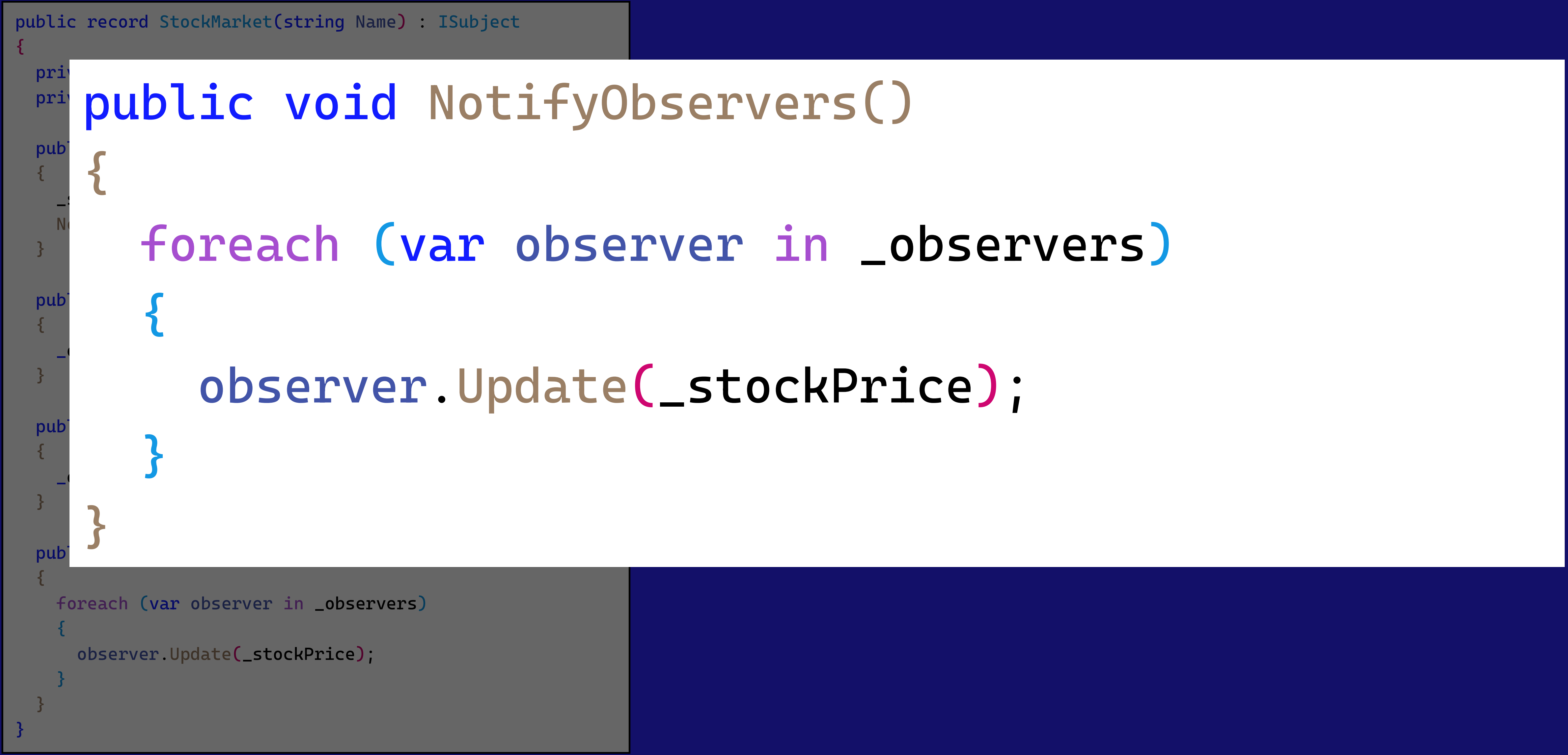
Concrete Subject



```
public record StockMarket(string Name) : ISubject
{
  pri
  pri

  pub
  {
    _s
    N
  }

  pub
  {
    _
  }

  pub
  {
    _
  }

  pub
  {
    foreach (var observer in _observers)
    {
      observer.Update(_stockPrice);
    }
  }
}
```

```
public void NotifyObservers()
{
  foreach (var observer in _observers)
  {
    observer.Update(_stockPrice);
  }
}
```

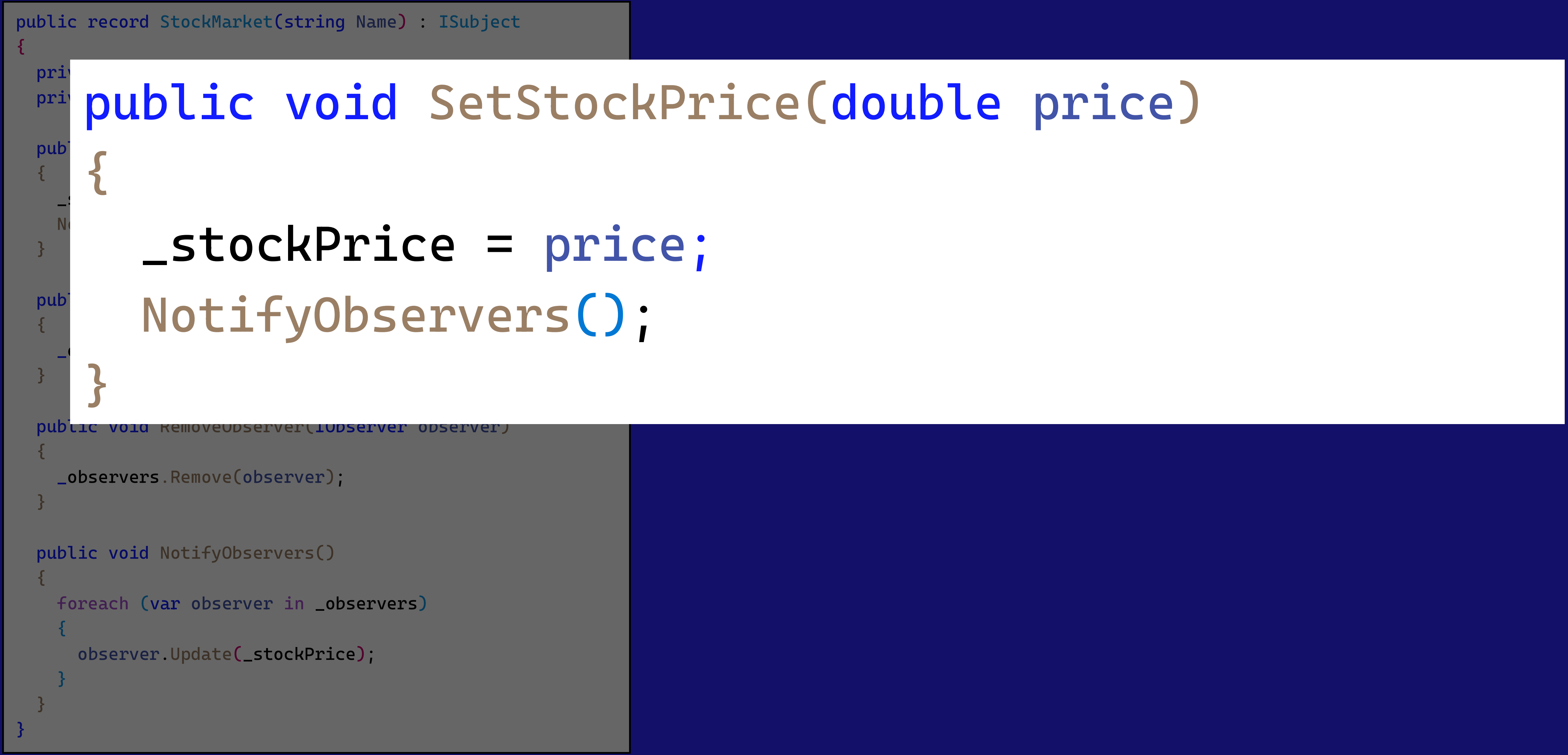
Concrete Subject



```
public record StockMarket(string Name) : ISubject
{
  pri
  pri

  pub
  {
    _
    N
  }

  pub
  {
    _
  }

  pub
  {
    _
  }

  pub
  {
    foreach (var observer in _observers)
    {
      observer.Update(_stockPrice);
    }
  }
}
```

```
public void NotifyObservers()
{
  foreach (var observer in _observers)
  {
    observer.Update(_stockPrice);
  }
}
```

Concrete Subject



```
public record StockMarket(string Name) : ISubject
{
  pri
  pri

  pub
  {
    _S
    N
  }

  pub
  {
    _
  }

  public void RemoveObserver(IObserver observer)
  {
    _observers.Remove(observer);
  }

  public void NotifyObservers()
  {
    foreach (var observer in _observers)
    {
      observer.Update(_stockPrice);
    }
  }
}
```

```
public void SetStockPrice(double price)
{
  _stockPrice = price;
  NotifyObservers();
}
```

Concrete Subject

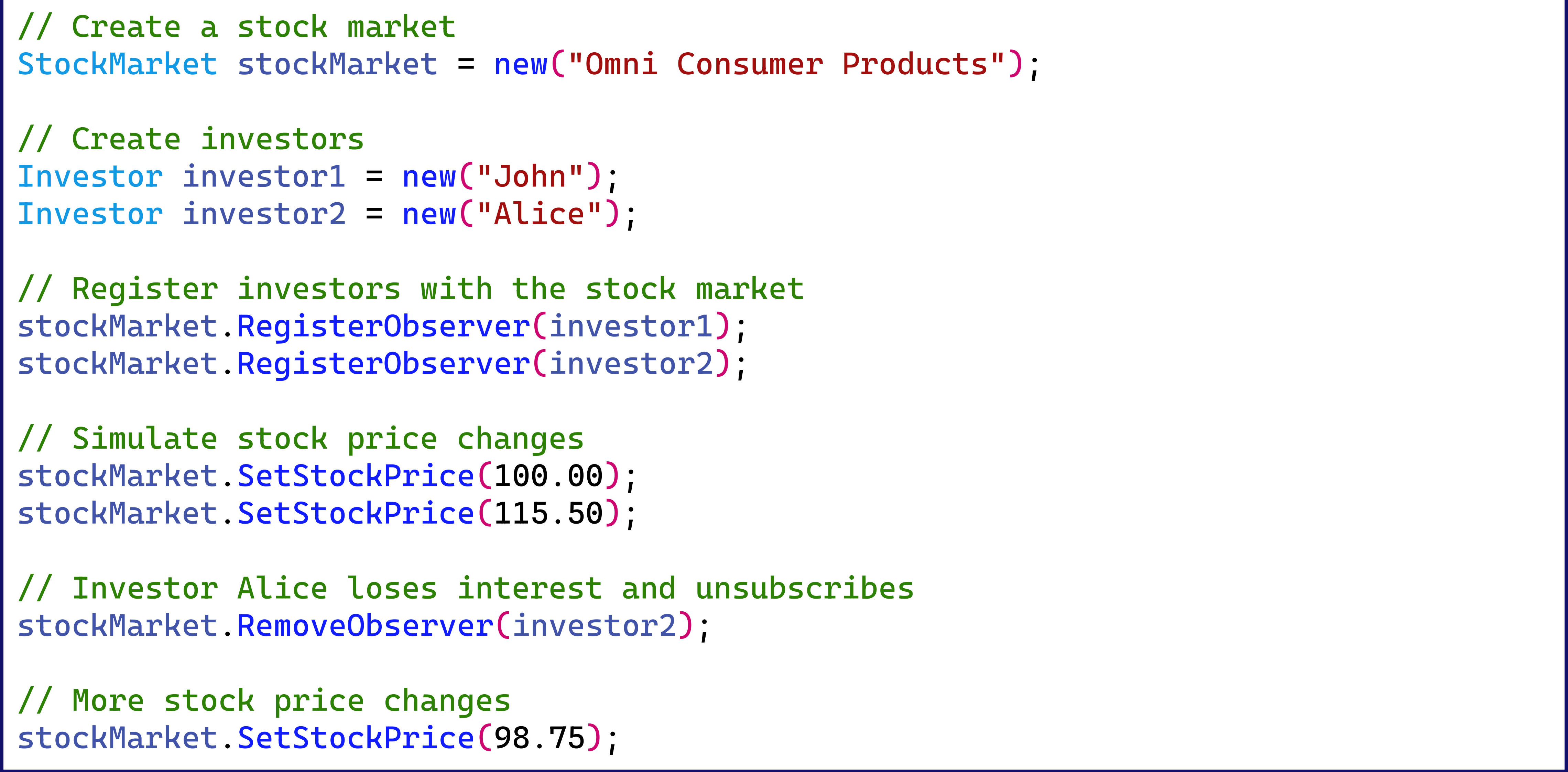
```
public record StockMarket(string Name) : ISubject
{
  pri
  pri

  pub
  {
    _s
    N
  }

  pub
  {
    _
  }

  public void RemoveObserver(IObserver observer)
  {
    _observers.Remove(observer);
  }

  public void NotifyObservers()
  {
    foreach (var observer in _observers)
    {
      observer.Update(_stockPrice);
    }
  }
}
```

```
public void SetStockPrice(double price)
{
  _stockPrice = price;
  NotifyObservers();
}
```

Concrete Observer

```
public record Investor(string Name) : IObserver
{
  public void Update(double stockPrice)
    => Console.WriteLine($"Stock price for {Name} is {stockPrice}");
}
```

Implementation



```
// Create a stock market
StockMarket stockMarket = new("Omni Consumer Products");

// Create investors
Investor investor1 = new("John");
Investor investor2 = new("Alice");

// Register investors with the stock market
stockMarket.RegisterObserver(investor1);
stockMarket.RegisterObserver(investor2);

// Simulate stock price changes
stockMarket.SetStockPrice(100.00);
stockMarket.SetStockPrice(115.50);

// Investor Alice loses interest and unsubscribes
stockMarket.RemoveObserver(investor2);

// More stock price changes
stockMarket.SetStockPrice(98.75);
```

Observer Pattern: The Good

Loose Coupling

Observer Pattern: The Good

Loose Coupling

Scalability

Observer Pattern: The Good

Loose Coupling

Scalability

Flexibility and Extensibility

Observer Pattern: The Good

Loose Coupling

Scalability

Flexibility and Extensibility

Reusability

Observer Pattern: The Good

Loose Coupling

Scalability

Flexibility and Extensibility

Reusability

Maintainability

Observer Pattern: The Good

Loose Coupling

Scalability

Flexibility and Extensibility

Reusability

Maintainability

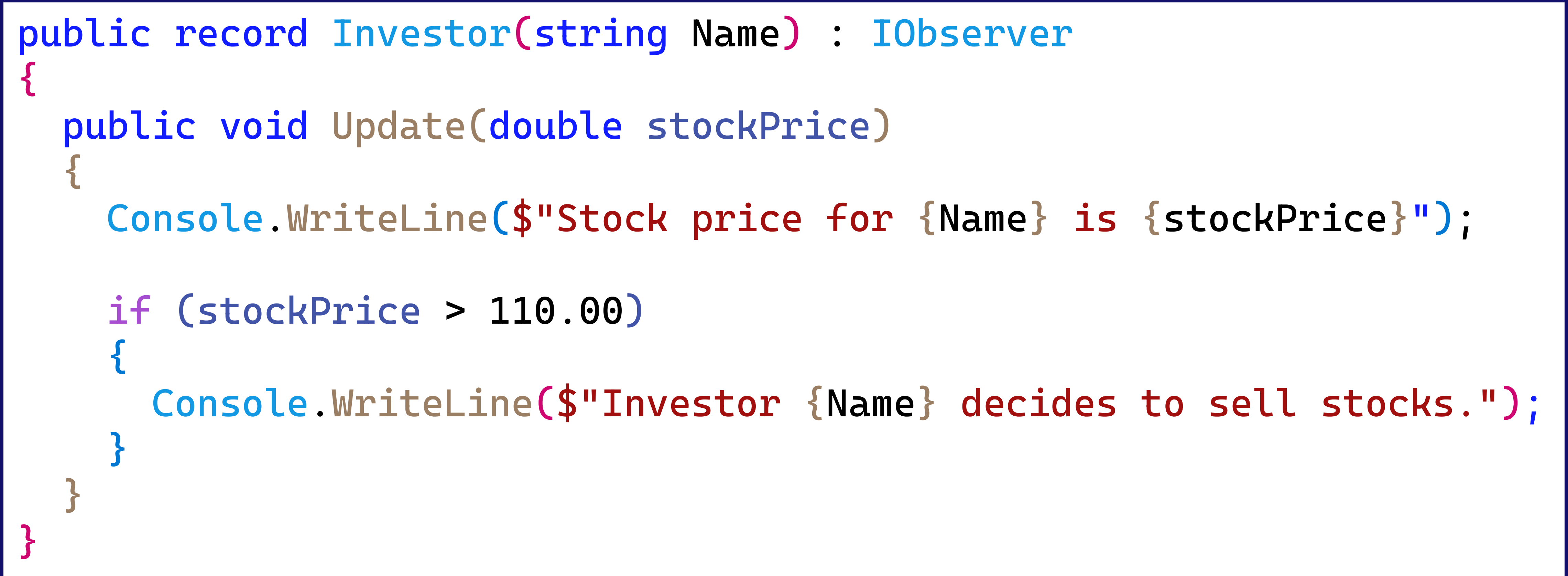
Dynamic Relationships

Observer Pattern: The Good

- Loose Coupling
- Scalability
- Flexibility and Extensibility
- Reusability
- Maintainability
- Dynamic Relationships

Demo: Observer Pattern Problems

Unintended Cascading Updates



```
public record Investor(string Name) : IObserver
{
  public void Update(double stockPrice)
  {
    Console.WriteLine($"Stock price for {Name} is {stockPrice}");

    if (stockPrice > 110.00)
    {
      Console.WriteLine($"Investor {Name} decides to sell stocks.");
    }
  }
}
```

Observer Pattern: The Bad

Performance

Observer Pattern: The Bad

Performance

Memory Leaks

Observer Pattern: The Bad

Performance

Memory Leaks

Ordering Dependencies

Observer Pattern: The Bad

- Performance
- Memory Leaks
- Ordering Dependencies
- Unintended Cascading Updates

Observer Pattern: The Bad

Performance

Memory Leaks

Ordering Dependencies

Unintended Cascading Updates

Security Concerns

Observer Pattern: The Bad

- Performance
- Memory Leaks
- Ordering Dependencies
- Unintended Cascading Updates
- Security Concerns
- Tight Coupling

Observer Pattern: The Bad

Performance

Memory Leaks

Ordering Dependencies

Unintended Cascading Updates

Security Concerns

Tight Coupling

Debugging Difficulty

Observer Pattern: The Bad

- Performance
- Memory Leaks
- Ordering Dependencies
- Unintended Cascading Updates
- Security Concerns
- Tight Coupling
- Debugging Difficulty

Alternatives/Modifications

- Event Aggregator Pattern

Alternatives/Modifications

- Event Aggregator Pattern
- **Reactive Extensions (Rx)**

Alternatives/Modifications

- Event Aggregator Pattern
- Reactive Extensions (Rx)
- **Mediator Pattern**

Alternatives/Modifications

- Event Aggregator Pattern
- Reactive Extensions (Rx)
- Mediator Pattern
- **Callback/Delegate Approach**

Alternatives/Modifications

- Event Aggregator Pattern
- Reactive Extensions (Rx)
- Mediator Pattern
- Callback/Delegate Approach
- **Message Queue Pattern**

Alternatives/Modifications

- Event Aggregator Pattern
- Reactive Extensions (Rx)
- Mediator Pattern
- Callback/Delegate Approach
- Message Queue Pattern
- **State Pattern**

Alternatives/Modifications

- Event Aggregator Pattern
- Reactive Extensions (Rx)
- Mediator Pattern
- Callback/Delegate Approach
- Message Queue Pattern
- State Pattern
- **Command Pattern**

Alternatives/Modifications

- **Event Aggregator Pattern**
- Reactive Extensions (Rx)
- **Mediator Pattern**
- Callback/Delegate Approach
- **Message Queue Pattern**
- State Pattern
- Command Pattern

Alternatives/Modifications

- Event Aggregator Pattern
- Reactive Extensions (Rx)
- Mediator Pattern
- Callback/Delegate Approach
- Message Queue Pattern
- State Pattern
- Command Pattern

Factory Pattern

Reevaluating Software Design Patterns

Key Components and Concepts

Factory Pattern

Factory Interface/ Abstract Class

Key Components and Concepts

Factory Pattern

Factory Interface/ Abstract Class

Concrete Factories

Key Components and Concepts

Factory Pattern

Factory Interface/ Abstract Class

Concrete Factories

Product Interface/ Abstract Class

Key Components and Concepts

Factory Pattern

Factory Interface/ Abstract Class

Concrete Factories

Product Interface/ Abstract Class

Concrete Products

Key Components and Concepts

Factory Pattern

Factory Interface/ Abstract Class

Concrete Factories

Product Interface/ Abstract Class

Concrete Products

Client

Key Components and Concepts

Factory Pattern

Factory Interface/ Abstract Class

Concrete Factories

Client

Product Interface/ Abstract Class

Concrete Products

Demo: Factory Pattern

Product

```
public interface IProduct
{
  void Display();
}

public class ConcreteProductA : IProduct
{
  public void Display() => Console.WriteLine("Concrete Product A");
}

public class ConcreteProductB : IProduct
{
  public void Display() => Console.WriteLine("Concrete Product B");
}
```

Product

```
public interface IProduct
{
  void Display();
}

public class ConcreteProductA : IProduct
{
  public void Display() => Console.WriteLine("Concrete Product A");
}

public class ConcreteProductB : IProduct
{
  public void Display() => Console.WriteLine("Concrete Product B");
}
```

Product

```
public interface IProduct
{
  void Display();
}
```

```
public class ConcreteProductA : IProduct
{
  public void Display() => Console.WriteLine("Concrete Product A");
}

public class ConcreteProductB : IProduct
{
  public void Display() => Console.WriteLine("Concrete Product B");
}
```

Factory

```
public interface IFactory
{
  IProduct CreateProduct();
}

public class ConcreteFactory : IFactory
{
  public IProduct CreateProduct()
  {
    return new ConcreteProductA();
  }
}
```

Client

```
IFactory factoryA = new ConcreteFactoryA();

IProduct productA = factoryA.CreateProduct();
productA.Display();

IProduct productB = factoryA.CreateProduct();
productB.Display();
```

Factory Pattern: The Good

Abstraction and Encapsulation

Factory Pattern: The Good

Abstraction and Encapsulation

Flexibility and Extensibility

Factory Pattern: The Good

- Abstraction and Encapsulation
- Flexibility and Extensibility
- Centralized Control

Factory Pattern: The Good

- Abstraction and Encapsulation
- Flexibility and Extensibility
- Centralized Control
- Code Maintenance

Factory Pattern: The Good

Abstraction and Encapsulation

Flexibility and Extensibility

Centralized Control

Code Maintenance

Code Readability

Factory Pattern: The Good

- Abstraction and Encapsulation
- Flexibility and Extensibility
- Centralized Control
- Code Maintenance
- Code Readability
- Dependency Inversion

Factory Pattern: The Good

- Abstraction and Encapsulation
- Flexibility and Extensibility
- Centralized Control
- Code Maintenance
- Code Readability
- Dependency Inversion
- **Separation of Concerns**

Factory Pattern: The Good

- Abstraction and Encapsulation
- Flexibility and Extensibility
- Centralized Control
- Code Maintenance
- Code Readability
- Dependency Inversion
- Separation of Concerns
- Consistency

Factory Pattern: The Good

- Abstraction and Encapsulation
- Flexibility and Extensibility
- Centralized Control
- Code Maintenance
- Code Readability
- Dependency Inversion
- Separation of Concerns
- Consistency

Factory Pattern: The Bad

Overhead

Factory Pattern: The Bad

Overhead

Excessive Abstraction

Factory Pattern: The Bad

Overhead

Excessive Abstraction

Tight Coupling

Factory Pattern: The Bad

- Overhead
- Excessive Abstraction
- Tight Coupling
- Factory Proliferation

Factory Pattern: The Bad

Overhead

Excessive Abstraction

Tight Coupling

Factory Proliferation

Complex Hierarchies

Factory Pattern: The Bad

Overhead

Excessive Abstraction

Tight Coupling

Factory Proliferation

Complex Hierarchies

Runtime Config Overhead

Factory Pattern: The Bad

- Overhead
- Excessive Abstraction
- Tight Coupling
- Factory Proliferation
- Complex Hierarchies
- Runtime Config Overhead
- Open/Closed Principle Violation

Factory Pattern: The Bad

- Overhead
- Excessive Abstraction
- Tight Coupling
- Factory Proliferation
- Complex Hierarchies
- Runtime Config Overhead
- Open/Closed Principle Violation
- Learning Curve

Factory Pattern: The Bad

- Overhead
- Excessive Abstraction
- Tight Coupling
- Factory Proliferation
- Complex Hierarchies
- Runtime Config Overhead
- Open/Closed Principle Violation
- Learning Curve

Alternatives to the Factory Pattern

- Direct Instantiation

Alternatives to the Factory Pattern

- Direct Instantiation
- **Builder Pattern**

Alternatives to the Factory Pattern

- Direct Instantiation
- Builder Pattern
- **Abstract Factory Pattern**

Alternatives to the Factory Pattern

- Direct Instantiation
- Builder Pattern
- **Abstract Factory Pattern**

Alternatives to the Factory Pattern

- Direct Instantiation
- Builder Pattern
- Abstract Factory Pattern
- **Static Factory Method**

Alternatives to the Factory Pattern

- Direct Instantiation
- Builder Pattern
- Abstract Factory Pattern
- Static Factory Method
- **Service Locator Pattern**

Alternatives to the Factory Pattern

- Direct Instantiation
- Builder Pattern
- Abstract Factory Pattern
- Static Factory Method
- Service Locator Pattern
- **Dependency Injection (DI)**

Alternatives to the Factory Pattern

- Direct Instantiation
- Builder Pattern
- Abstract Factory Pattern
- Static Factory Method
- Service Locator Pattern
- Dependency Injection (DI)
- **Strategy Pattern**

Alternatives to the Factory Pattern

- **Direct Instantiation**
- Builder Pattern
- Abstract Factory Pattern
- **Static Factory Method**
- Service Locator Pattern
- Dependency Injection (DI)
- Strategy Pattern

Importance of Context

Reevaluating Software Design Patterns

Importance of Context

Problem Suitability

Importance of Context

Problem Suitability

Project Requirements

Importance of Context

- Problem Suitability
- Project Requirements
- Team Expertise

Importance of Context

- Problem Suitability
- Project Requirements
- Team Expertise
- Technology Stack

Importance of Context

- Problem Suitability
- Project Requirements
- Team Expertise
- Technology Stack
- System Evolution

Importance of Context

- Problem Suitability
- Project Requirements
- Team Expertise
- Technology Stack
- System Evolution
- Performance Considerations

Importance of Context

- Problem Suitability
- Project Requirements
- Team Expertise
- Technology Stack
- System Evolution
- Performance Considerations
- **Trade-offs and Constraints**

Importance of Context

- Problem Suitability
- Project Requirements
- Team Expertise
- Technology Stack
- System Evolution
- Performance Considerations
- Trade-offs and Constraints

Thank You

chadgreen@chadgreen.com

TaleLearnCode

ChadGreen.com

ChadGreen & TaleLearnCode

ChadwickEGreen

Microsoft® Most Valuable Professional MVP