# Лабораторная работа № 1 События

**Цель лабораторной работы**

Познакомиться с основами программирования событий.

**Теоретический материал**

**Делегаты**

Делегатыпредставляют объекты, которые указывают на методы. Методы, на которые ссылаются делегаты, должны иметь те же параметры и тот же тип возвращаемого значения.

Для объявления делегата используется ключевое слово *delegate*, после которого идет возвращаемый тип, название и параметры:

delegate int Operation(int x, int y);

//здесь делегат ссылается на функцию, которая в //качестве параметров принимает два значения типа int и //возвращает некоторое число

delegate void GetMessage();

Для использования делегата надо создать его объект с помощью конструктора, в который мы передаем адрес метода, вызываемого делегатом. Чтобы вызвать метод, на который указывает делегат, надо использовать его метод *Invoke* (или *BeginInvoke/EndInvoke* при асинхронном режиме выполнения делегатов).

Пример с использованием делегата:

class Program

{

// 1. Объявляем делегат

delegate int Operation(int x, int y);

static void Main(string[] args)

{

// присваивание адреса метода через конструктор

// 2. Создаем переменную делегата

Operation del = new Operation(Add);

// 3. делегат указывает на метод Add

// 4. Вызываем метод

int result = del.Invoke(4, 5);

Console.WriteLine(result);

del = Multiply;

// теперь делегат указывает на метод Multiply

result = del.Invoke(4, 5);

Console.WriteLine(result);

Console.Read();

}

private static int Add(int x, int y)

{

return x + y;

}

private static int Multiply(int x, int y)

{

return x \* y;

}

}

Здесь объявляется делегат Operation, возвращающий целочисленное значение и принимающий два параметра, также целочисленные. Методы (Add и Multiply), на которые указывает созданный экземпляр делегата, определяются с использованием конструктора.

Метод Invoke() при вызове делегата можно опустить и использовать сокращенную форму:

del = Multiply;

// теперь делегат указывает на метод Multiply

result = del.Invoke(4, 5);

Использование делегатов обусловлено тем, что не всегда есть доступ к коду классов. Например, часть классов может создаваться и компилироваться одним человеком, который не будет знать, как эти классы будут использоваться. А использовать эти классы будет другой разработчик.

**Анонимные методы**

Анонимные методы используются для создания экземпляров делегатов. Определение анонимных методов начинается с ключевого слова *delegate*, после которого идет в скобках список параметров и тело метода в фигурных скобках:

delegate (параметры)

{

// инструкции

}

Пример:

class Program

{

delegate void MessageHandler(string message);

static void Main(string[] args)

{

MessageHandler handler = delegate (string mes)

{

Console.WriteLine(mes);

};

handler("hello world!");

Console.Read();

}

}

Анонимный метод не может существовать сам по себе, он используется для инициализации экземпляра делегата, как в данном случае переменная handler представляет анонимный метод. И через эту переменную делегата можно вызвать данный анонимный метод.

В отличие от блока методов или условных и циклических конструкций, блок анонимных методов должен заканчиваться точкой с запятой после закрывающей фигурной скобки.

Если анонимный метод использует параметры, то они должны соответствовать параметрам делегата. Если для анонимного метода не требуется параметров, то скобки с параметрами опускаются. При этом даже если делегат принимает несколько параметров, то в анонимном методе можно вовсе опустить параметры.

Анонимные методы используются, когда надо определить однократное действие, которое не имеет много инструкций и нигде больше не используется.

Также, как и обычные методы, анонимные могут возвращать результат:

class Program

{

delegate int Operation(int x, int y);

static void Main(string[] args)

{

Operation operation = delegate (int x, int y)

{

return x + y;

};

int d = operation(4, 5);

Console.WriteLine(d); // 9

Console.Read();

}

}

**Лямбда-выражения**

Лямбда-выражения представляют упрощенную запись анонимных методов. Лямбда-выражения позволяют создать емкие лаконичные методы, которые могут возвращать некоторое значение и которые можно передать в качестве параметров в другие методы.

Синтаксис лямбда-выражений:

слева от лямбда-оператора => определяется список параметров, а справа блок выражений, использующий эти параметры:

(список\_параметров) => выражение.

Например:

class Program

{

delegate int Operation(int x, int y);

static void Main(string[] args)

{

Operation operation = (x, y) => x + y;

Console.WriteLine(operation(10, 20)); // 30

Console.WriteLine(operation(40, 20)); // 60

Console.Read();

}

}

Здесь x, y => x + y; – лямбда-выражение, x и y – параметры, x + y – выражение.

При этом не надо указывать тип параметров, а при возвращении результата не надо использовать оператор return.

Возвращаемое значение лямбда-выражений должно быть тем же, что и у делегата. Если лямбда-выражение принимает один параметр, то скобки вокруг параметра можно опустить, если параметров нет – ставятся пустые скобки.

Также бывает, что лямбда-выражение не возвращает никакого значения, и вызывает другие методы:

class Program

{

delegate void Hello(); // делегат без параметров

static void Main(string[] args)

{

Hello message = () => Show\_Message();

message();

}

private static void Show\_Message()

{

Console.WriteLine("Вызов сообщения через делегат");

}

}

Одним из частых примеров использования лямбда-выражений является обработка событий. Как и делегаты, лямбда-выражения можно передавать в качестве аргументов методу для тех параметров, которые представляют делегат, что довольно удобно.

**События**

События сигнализируют системе о том, что произошло определенное действие. События объявляются в классе с помощью ключевого слова event, после которого идет название делегата.

// Объявляем делегат

public delegate void AccountStateHandler(

string message);

// Событие, возникающее при выводе денег

public event AccountStateHandler Withdrawn;

Связь с делегатом означает, что метод, обрабатывающий данное событие, должен принимать те же параметры, что и делегат, и возвращать тот же тип, что и делегат. Рассмотрим пример:

class Account

{

// Объявляем делегат

public delegate void AccountStateHandler(string message);

// Событие, возникающее при выводе денег

public event AccountStateHandler Withdrawn;

// Событие, возникающее при добавление на счет

public event AccountStateHandler Added;

int \_sum; // Переменная для хранения суммы

public Account(int sum)

{

\_sum = sum;

}

public int CurrentSum

{

get { return \_sum; }

}

public void Put(int sum)

{

\_sum += sum;

if (Added != null)

Added($"На счет поступило {sum}");

}

public void Withdraw(int sum)

{

if (sum <= \_sum)

{

\_sum -= sum;

if (Withdrawn != null) Withdrawn($"Сумма {sum} снята со счета");

}

else

{

if (Withdrawn != null) Withdrawn("Недостаточно денег на счете");

}

}

}

События Withdrawn и Added объявлены как экземпляры делегата AccountStateHandler, поэтому для их обработки потребуется метод, принимающий строку в качестве параметра. Затем в методах Put и Withdraw эти события вызываются. Перед вызовом мы проверяем, закреплены ли за этими событиями обработчики. Так как эти события представляют делегат AccountStateHandler, принимающий в качестве параметра строку, то и при вызове событий мы передаем в них строку.

Для прикрепления обработчика события к определенному событию используется операция += и, соответственно, для открепления – операция -=:

событие += метод\_обработчика\_события.

Метод обработчика должен иметь такие же параметры, как и делегат события, и возвращать тот же тип:

class Program

{

static void Main(string[] args)

{

Account account = new Account(200);

// Добавляем обработчики события

account.Added += Show\_Message;

account.Withdrawn += Show\_Message;

account.Withdraw(100);

// Удаляем обработчик события

account.Withdrawn -= Show\_Message;

account.Withdraw(50);

account.Put(150);

Console.ReadLine();

}

private static void Show\_Message(string message)

{

Console.WriteLine(message);

}

}

Оба способа прикрепления обработчиков равноценны:

account.Added += Show\_Message;

account.Added += new

Account.AccountStateHandler(Show\_Message);

**Задание на лабораторную работу**

1. Разработать библиотеку классов согласно варианту.
2. Формализовать предметную область в виде классов со свойствами, полями, методами, делегатами и событиями.
3. В этом же решении создать проект «Консольное приложение».
4. Подключить библиотеку классов к консольному приложению.
5. В консольном приложении продемонстрировать работу с разработанными классами.
6. Написать XML комментарии (с помощью ///) к классам и методам.
7. Подготовить тестовые сценарии для проверки работы приложения.

**Варианты заданий на лабораторную работу**

**1. Система управления библиотекой**

1. **Добавление новой книги** – регистрация книги в каталоге.
2. **Выдача книги читателю** – фиксация факта выдачи и срока возврата.
3. **Возврат книги** – подтверждение сдачи книги в библиотеку.
4. **Просрочка возврата** – уведомление о нарушении сроков.
5. **Списание книги** – удаление из фонда (утеря, повреждение).

**2. Система управления заказами в ресторане**

1. **Создание заказа** – оформление нового заказа клиентом.
2. **Изменение заказа** – добавление/удаление блюд.
3. **Перевод заказа на кухню** – передача поварам на исполнение.
4. **Завершение заказа** – отметка о выполнении.
5. **Оплата заказа** – фиксация платежа (наличные/карта).

**3. Управление проектами**

1. **Создание проекта** – добавление нового проекта в систему.
2. **Назначение задачи исполнителю** – распределение работы в команде.
3. **Изменение статуса задачи** (в работе, на проверке, завершена).
4. **Дедлайн просрочен** – уведомление о невыполнении в срок.
5. **Закрытие проекта** – подведение итогов и архивация.

**4. Система управления учебным процессом**

1. **Зачисление студента** – добавление записи в текстовый файл.
2. **Запись на курс** – выбор дисциплины студентом.
3. **Проведение лекции/семинара** – отметка о посещении.
4. **Сдача экзамена/зачета** – фиксация оценки.
5. **Отчисление студента** – завершение обучения.

**5. Система учета финансов**

1. **Поступление средств** – зачисление денег на счет.
2. **Списание средств** – оплата расходов.
3. **Формирование отчета** – автоматическая генерация финансового отчета.
4. **Обнаружение перерасхода** – уведомление о превышении бюджета.
5. **Конвертация валюты** – изменение курса и пересчет баланса.

**6. Система бронирования отелей**

1. **Создание брони** – выбор номера и дат проживания.
2. **Отмена бронирования** – возврат средств (если предусмотрено).
3. **Заселение гостя** – подтверждение прибытия.
4. **Выселение гостя** – завершение проживания.
5. **Изменение тарифа** – корректировка стоимости номера.

**7. Система управления мультимедиа**

1. **Загрузка файла** – добавление медиа (фото, видео, аудио).
2. **Редактирование метаданных** – изменение названия, тегов.
3. **Удаление файла** – очистка архива.
4. **Публикация контента** – размещение на платформе.
5. **Превышение квоты хранилища** – уведомление о нехватке места.

**8. Система управления спортивной командой**

1. **Добавление игрока** – подписание контракта.
2. **Травма игрока** – внесение в список травмированных.
3. **Замена в матче** – изменение состава во время игры.
4. **Назначение тренера** – выбор нового наставника.
5. **Завершение сезона** – подведение статистики.

**9. Система управления событиями**

1. **Создание мероприятия** – определение даты, места, участников.
2. **Регистрация участника** – подтверждение участия.
3. **Отмена события** – уведомление всех заинтересованных.
4. **Изменение программы** – корректировка расписания.
5. **Завершение события** – фиксация результатов и отзывов.

**10. Система управления складом**

1. **Поступление товара** – добавление на склад.
2. **Продажа товара** – списание со склада.
3. **Инвентаризация** – сверка фактического наличия с учетными данными.
4. **Остаток ниже минимума** – уведомление о необходимости закупки.
5. **Просроченный товар** – списание испорченной продукции.

**11. Система управления контактами**

1. **Добавление нового контакта** – внесение данных о человеке или организации
2. **Редактирование контакта** – обновление информации (телефон, email и др.)
3. **Удаление контакта** – удаление записи
4. **Группировка контактов** – добавление в категорию (друзья, коллеги и т. д.)
5. **Экспорт контактов** – выгрузка данных в файл (CSV, vCard)

**12. Система управления задачами для личного использования**

1. **Создание задачи** – добавление нового пункта в список дел
2. **Изменение статуса задачи** (например: "в работе", "завершено")
3. **Назначение срока выполнения** – установка дедлайна
4. **Добавление метки/категории** – сортировка задач по тегам
5. **Напоминание о задаче** – уведомление о приближении дедлайна

**13. Система управления подписками**

1. **Оформление подписки** – выбор тарифа и оплата
2. **Продление подписки** – автоматическое или ручное обновление
3. **Отмена подписки** – прекращение доступа к услуге
4. **Изменение тарифного плана** – переход на другой пакет услуг
5. **Уведомление об истечении срока** – предупреждение перед автопродлением

**14. Система управления резюме**

1. **Создание резюме** – добавление информации о навыках и опыте
2. **Обновление данных** – редактирование разделов (образование, работа)
3. **Публикация резюме** – размещение на платформе для поиска работы
4. **Скачивание резюме** – экспорт в PDF или DOCX
5. **Отклик на вакансию** – отправка резюме работодателю

**15. Система управления опросами**

1. **Создание опроса** – добавление вопросов и вариантов ответов
2. **Запуск опроса** – открытие доступа для участников
3. **Завершение опроса** – закрытие голосования
4. **Анализ результатов** – подсчет голосов и генерация отчета
5. **Публикация итогов** – размещение результатов в открытом доступе

**16. Система управления благотворительными проектами**

1. **Создание проекта** – добавление описания и целей сбора
2. **Пожертвование** – фиксация перевода средств
3. **Достижение цели** – уведомление о завершении сбора
4. **Отчет о расходах** – публикация информации, куда пошли деньги
5. **Закрытие проекта** – подведение итогов и благодарности

**17. Система управления фитнес-программами**

1. **Создание тренировки** – добавление упражнений и плана
2. **Назначение программы пользователю** – выбор персонального плана
3. **Отметка о выполнении** – фиксация завершенной тренировки
4. **Корректировка нагрузки** – изменение сложности упражнений
5. **Анализ прогресса** – сравнение результатов за период

**18. Система управления проектами** *(уточненная версия)*

1. **Создание проекта** – определение целей и сроков
2. **Назначение ответственного** – выбор руководителя проекта
3. **Изменение статуса** (планирование, в работе, завершен)
4. **Риск-менеджмент** – фиксация потенциальных проблем
5. **Подведение итогов** – отчет о выполнении задач

**19. Система управления складом** *(расширенная версия)*

1. **Поступление товара** – добавление в базу с указанием поставщика
2. **Резервирование товара** – временное удержание для заказа
3. **Перемещение между складами** – изменение места хранения
4. **Уценка товара** – изменение стоимости (например, из-за повреждения)
5. **Автоматический заказ** – уведомление о необходимости пополнения запасов

**20. Система управления складом**

1. **Поступление товара** – добавление новой партии товара на склад с указанием поставщика, количества и даты
2. **Резервирование товара** – временное удержание позиций для конкретного заказа или клиента
3. **Перемещение между складами** – передача товара в другой филиал или зону хранения
4. **Инвентаризация** – сверка фактического наличия товара с учетными данными
5. **Списание товара** – удаление испорченных, утерянных или просроченных позиций

**21. Система управления ресторанами**

1. **Добавление нового блюда в меню** – внесение информации о составе и цене
2. **Бронирование столика** – фиксация даты, времени и количества гостей
3. **Заказ доставки** – оформление заказа с указанием адреса и способа оплаты
4. **Отзыв о ресторане** – публикация оценки и комментария клиента
5. **Изменение режима работы** – обновление графика открытия/закрытия

**22. Система управления мероприятиями**

1. **Создание мероприятия** – определение названия, даты и места проведения
2. **Регистрация участника** – подтверждение участия с указанием контактов
3. **Изменение программы** – корректировка списка выступающих или расписания
4. **Отмена мероприятия** – уведомление участников и возврат билетов
5. **Публикация фотоотчета** – размещение материалов после завершения

**23. Система управления финансами**

1. **Пополнение счета** – зачисление средств на баланс
2. **Списание средств** – оплата услуг или товаров
3. **Формирование бюджета** – планирование доходов и расходов
4. **Автоматическое начисление процентов** – капитализация вклада
5. **Генерация налогового отчета** – подготовка документа для ФНС

**24. Система управления путешествиями**

1. **Бронирование билетов** – выбор рейса/поезда и оплата
2. **Регистрация в отеле** – подтверждение дат проживания и номера
3. **Изменение маршрута** – корректировка точек посещения
4. **Отмена поездки** – возврат средств и уведомление сервисов
5. **Добавление отзыва** – оценка отеля, авиакомпании или экскурсии

**25. Система управления медицинскими записями**

1. **Запись на прием** – выбор врача, даты и времени
2. **Добавление диагноза** – внесение результатов осмотра
3. **Назначение лечения** – выписка рецепта или процедур
4. **Перенос приема** – изменение даты визита к врачу
5. **Экспорт медицинской карты** – передача данных в другую клинику

**26. Система управления событиями для некоммерческих организаций**

1. **Подача заявки на волонтерство** – регистрация желающего помочь
2. **Создание благотворительного сбора** – определение цели и суммы
3. **Публикация отчета** – размещение информации о расходовании средств
4. **Организация мероприятия** – анонс акции или фандрайзинга
5. **Благодарность участникам** – отправка сертификатов или писем

**Пример разработки приложения**

**Система управления заказами в интернет-магазине**

**Описание**: Эта система предназначена для управления заказами в интернет-магазине, включая управление товарами, клиентами и статусами заказов.

**Шаг 1. Создание проекта**

1. Откройте **Visual Studio**.
2. Нажмите **"Создать новый проект"**.
3. Выберите **"Консольное приложение"**.
4. Назовите проект (например, OnlineShopManagement) и нажмите **"Создать"**.

**Шаг 2. Создание перечисления**OrderStatus

В файле Program.cs (или создайте новый файл OrderStatus.cs) добавьте:

// Enum для статуса заказа

public enum OrderStatus

{

Processing,

Shipped,

Delivered,

Canceled

}

**Зачем это нужно?** Это список возможных статусов заказа, которые будут использоваться в классе Order.

**Шаг 3. Создание класса**Product**(Товар)**

1. Создайте новый файл Product.cs (или добавьте в Program.cs).
2. Добавьте свойства:
   * ProductID (уникальный код товара)
   * Name (название)
   * Price (цена)
   * StockQuantity (количество на складе)
3. Реализуйте методы:
   * AddStock() – увеличивает количество товара.
   * RemoveStock() – уменьшает количество (если товара достаточно).
   * CalculateTotalPrice() – вычисляет стоимость для указанного количества.
   * GetProductSummary() – возвращает строку с информацией о товаре.
4. **Проверка:**

Убедитесь, что класс реализует интерфейсы ICloneable и IEquatable<Product>.

namespace ProductsLibrary;

public record ProductStockAddedEventArgs(int Quantity);

public delegate void ProductStockAdded(object sender, ProductStockAddedEventArgs args);

public record ProductStockRemovedEventArgs(int Quantity);

public delegate void ProductStockRemoved(object sender, ProductStockRemovedEventArgs args);

// Класс для представления продукта

public class Product : ICloneable, IEquatable<Product>

{

public event ProductStockAdded? ProductStockAdded;

public event ProductStockRemoved? ProductStockRemoved;

public string ProductID { get; private set; }

public string Name { get; private set; }

public double Price { get; private set; }

public int StockQuantity { get; private set; }

public Product(string productId, string name, double price, int stockQuantity)

{

ProductID = productId;

Name = name;

Price = price;

StockQuantity = stockQuantity;

}

public void AddStock(int quantity)

{

StockQuantity += quantity;

ProductStockAdded?.Invoke(this, new ProductStockAddedEventArgs(quantity));

}

public bool RemoveStock(int quantity)

{

if (StockQuantity >= quantity)

{

StockQuantity -= quantity;

ProductStockRemoved?.Invoke(this, new ProductStockRemovedEventArgs(quantity));

return true;

}

else

{

ProductStockRemoved?.Invoke(this, new ProductStockRemovedEventArgs(quantity));

return false;

}

}

public double CalculateTotalPrice(int quantity)

{

return Price \* quantity;

}

public string GetProductSummary()

{

return $"{Name} (ID: {ProductID}) - Цена: {Price}, Остаток: {StockQuantity}";

}

public override string ToString()

{

return GetProductSummary();

}

public object Clone()

{

return MemberwiseClone();

}

public bool Equals(Product? other)

{

if (other == null)

return false;

return other.ProductID == ProductID;

}

public override bool Equals(object obj)

{

return Equals(obj as Product);

}

public override int GetHashCode()

{

return ProductID.GetHashCode();

}

}

**Шаг 4. Создание класса**Customer**(Клиент)**

1. Создайте файл Customer.cs.
2. Добавьте свойства:
   * CustomerID (уникальный код клиента)
   * Name (имя)
   * Email (email, с проверкой формата)
   * Address (адрес, с проверкой)
   * Orders (список заказов клиента)
3. Реализуйте методы:
   * SetEmail() – проверяет, содержит ли email символ @.
   * SetAddress() – проверяет, что адрес не пустой.
   * AddOrder() – добавляет заказ в список.
   * HasOrders() – проверяет, есть ли заказы у клиента.
4. **Проверка:**

Убедитесь, что класс реализует ICloneable и IEquatable<Customer>.

namespace ProductsLibrary;

public record CustomerAddedOrderEventArgs(Order Order);

public delegate void CustomerAddedOrder(object sender, CustomerAddedOrderEventArgs args);

// Класс для представления клиента

public class Customer : ICloneable, IEquatable<Customer>

{

public event CustomerAddedOrder? CustomerAddedOrder;

public string CustomerID { get; private set; }

public string Name { get; private set; }

public IReadOnlyList<Order> Orders => orders.AsReadOnly();

private string email;

private string address;

private List<Order> orders;

public Customer(string customerId, string name, string email, string address)

{

CustomerID = customerId;

Name = name;

SetEmail(email);

SetAddress(address);

orders = new List<Order>();

}

public void SetEmail(string newEmail)

{

if (IsValidEmail(newEmail))

{

email = newEmail;

}

else

{

throw new ArgumentException("Неверный формат электронной почты.");

}

}

public void SetAddress(string newAddress)

{

if (IsValidAddress(newAddress))

{

address = newAddress;

}

else

{

throw new ArgumentException("Неверный формат адреса.");

}

}

public string GetCustomerInfo()

{

return $"Клиент: {Name} (ID: {CustomerID}), Email: {email}, Адрес: {address}";

}

public override string ToString()

{

return GetCustomerInfo();

}

public bool HasOrders()

{

return orders.Count > 0; // Проверка на наличие заказов

}

public void AddOrder(Order order)

{

orders.Add(order);

CustomerAddedOrder?.Invoke(this, new CustomerAddedOrderEventArgs(order));

}

public void AddOrders(IEnumerable<Order> orders)

{

foreach (var order in orders)

{

AddOrder(order);

}

}

private bool IsValidEmail(string email)

{

// Проверка на наличие символа "@" и точки "."

return email.Contains("@") && email.IndexOf('.') > email.IndexOf('@') + 1;

}

private bool IsValidAddress(string address)

{

// Проверка, что адрес не пустой и содержит хотя бы 5 символов

return !string.IsNullOrWhiteSpace(address) && address.Length >= 5;

}

public object Clone()

{

var customer = new Customer(CustomerID, Name, email, address);

var orders = new List<Order>();

foreach (var order in Orders)

{

customer.AddOrder((Order)order.Clone());

}

return customer;

}

public bool Equals(Customer? other)

{

if (other == null)

return false;

return other.CustomerID == CustomerID;

}

public override bool Equals(object obj)

{

return Equals(obj as Customer);

}

public override int GetHashCode()

{

return CustomerID.GetHashCode();

}

}

**Шаг 5. Создание класса**Order**(Заказ)**

1. Создайте файл Order.cs.
2. Добавьте свойства:
   * OrderID (уникальный код заказа)
   * Customer (клиент, сделавший заказ)
   * Products (список товаров в заказе)
   * OrderDate (дата создания)
   * Status (статус из OrderStatus)
3. Реализуйте методы:
   * AddProduct() – добавляет товар в заказ (уменьшает остаток на складе).
   * UpdateStatus() – меняет статус заказа.
   * CalculateTotalOrderValue() – считает общую сумму заказа.
   * GetOrderSummary() – возвращает строку с информацией о заказе.
4. **Проверка:**

Убедитесь, что класс реализует ICloneable, IEquatable<Order> и IEnumerable<Product>.

using System.Collections;

namespace ProductsLibrary;

public record OrderStatusUpdatedEventArgs(OrderStatus Old, OrderStatus New);

public delegate void OrderStatusUpdated(object sender, OrderStatusUpdatedEventArgs args);

public record ProductAddedToOrderEventArgs(Product Product);

public delegate void ProductAddedToOrder(object sender, ProductAddedToOrderEventArgs args);

// Класс для представления заказа

public class Order : ICloneable, IEquatable<Order>, IEnumerable<Product>

{

public event ProductAddedToOrder? ProductAddedToOrder;

public event OrderStatusUpdated? OrderStatusUpdated;

public string OrderID { get; private set; }

public Customer Customer { get; private set; }

public IReadOnlyList<Product> Products => products.AsReadOnly();

public DateTime OrderDate { get; private set; }

public OrderStatus Status { get; private set; }

private List<Product> products;

public Order(string orderId, Customer customer)

{

OrderID = orderId;

Customer = customer;

products = new List<Product>();

OrderDate = DateTime.Now;

Status = OrderStatus.Processing;

}

public bool AddProduct(Product product)

{

if (product.RemoveStock(1)) // Уменьшаем количество товара на складе

{

products.Add(product);

ProductAddedToOrder?.Invoke(this, new ProductAddedToOrderEventArgs(product));

return true;

}

return false;

}

public void UpdateStatus(OrderStatus status)

{

var old = Status;

Status = status;

OrderStatusUpdated?.Invoke(this, new OrderStatusUpdatedEventArgs(old, Status));

}

public double CalculateTotalOrderValue()

{

double total = 0;

foreach (var product in Products)

{

total += product.Price; // Предполагаем, что количество каждого продукта в заказе равно 1

}

return total;

}

public string GetOrderSummary()

{

string productDetails = string.Join(", ", products.ConvertAll(p => p.GetProductSummary()));

return $"Заказ ID: {OrderID}, " +

$"Клиент: {Customer.Name}, " +

$"Товары: {productDetails}, " +

$"Дата: {OrderDate}, " +

$"Статус: {Status}, " +

$"Общая сумма: {CalculateTotalOrderValue()}";

}

public override string ToString()

{

return GetOrderSummary();

}

public object Clone()

{

var order = new Order(OrderID, (Customer)Customer.Clone());

var products = new List<Product>();

foreach (var product in Products)

{

order.AddProduct((Product)product.Clone());

}

order.OrderDate = OrderDate;

order.Status = Status;

return order;

}

public bool Equals(Order? other)

{

if (other == null)

return false;

return other.OrderID == OrderID;

}

public override bool Equals(object obj)

{

return Equals(obj as Order);

}

public override int GetHashCode()

{

return OrderID.GetHashCode();

}

public IEnumerator<Product> GetEnumerator()

{

return Products.GetEnumerator();

}

IEnumerator IEnumerable.GetEnumerator()

{

return ((IEnumerable)Products).GetEnumerator();

}

}

**Шаг 6. Настройка главного класса**Program

1. В Program.cs создайте:
   * Несколько товаров (Product).
   * Клиента (Customer).
   * Заказ (Order).
2. Добавьте товары в заказ (order.AddProduct()).
3. Обновите статус заказа (order.UpdateStatus()).
4. Добавьте заказ клиенту (customer.AddOrder()).
5. Выведите информацию:
   * О клиенте (customer.GetCustomerInfo()).
   * О заказе (order.GetOrderSummary()).

using ProductsLibrary;

using System;

public class Program

{

public static void Main(string[] args)

{

Console.WriteLine("=== Тестирование событий ===");

// Создаем продукты

Product product1 = new Product("P001", "Ноутбук", 799.99, 10);

Product product2 = new Product("P002", "Смартфон", 499.99, 20);

// Подписываемся на события Product

product1.ProductStockAdded += ProductStockAddedHandler;

product1.ProductStockRemoved += ProductStockRemovedHandler;

product2.ProductStockAdded += ProductStockAddedHandler;

product2.ProductStockRemoved += ProductStockRemovedHandler;

Console.WriteLine("\n=== Тестирование событий Product ===");

product1.AddStock(5); // Должно вызвать ProductStockAdded

product1.RemoveStock(3); // Должно вызвать ProductStockRemoved

product1.RemoveStock(20); // Должно вызвать ProductStockRemoved (но не удастся)

// Создаем клиента

Customer customer = new Customer("C001", "Иван Иванов", "ivan@example.com", "Москва, ул. Пушкина, д. 1");

// Подписываемся на события Customer

customer.CustomerAddedOrder += CustomerAddedOrderHandler;

// Создаем заказ

Order order = new Order("O001", customer);

// Подписываемся на события Order

order.ProductAddedToOrder += ProductAddedToOrderHandler;

order.OrderStatusUpdated += OrderStatusUpdatedHandler;

Console.WriteLine("\n=== Тестирование событий Order ===");

// Добавляем продукты в заказ

if (order.AddProduct(product1)) // Должно вызвать ProductAddedToOrder и ProductStockRemoved

{

Console.WriteLine($"Продукт '{product1.Name}' добавлен в заказ.");

}

if (order.AddProduct(product2)) // Должно вызвать ProductAddedToOrder и ProductStockRemoved

{

Console.WriteLine($"Продукт '{product2.Name}' добавлен в заказ.");

}

// Обновляем статус заказа (должно вызвать OrderStatusUpdated)

order.UpdateStatus(OrderStatus.Shipped);

Console.WriteLine("\n=== Тестирование событий Customer ===");

// Добавляем заказ к клиенту (должно вызвать CustomerAddedOrder)

customer.AddOrder(order);

// Выводим информацию

Console.WriteLine("\n=== Итоговая информация ===");

Console.WriteLine(customer.GetCustomerInfo());

Console.WriteLine($"У клиента заказы: {customer.HasOrders()}");

Console.WriteLine(order.GetOrderSummary());

}

// Обработчики событий

private static void ProductStockAddedHandler(object sender, ProductStockAddedEventArgs args)

{

var product = (Product)sender;

Console.WriteLine($"[Событие] Количество товара '{product.Name}' увеличено на {args.Quantity}. Новый остаток: {product.StockQuantity}");

}

private static void ProductStockRemovedHandler(object sender, ProductStockRemovedEventArgs args)

{

var product = (Product)sender;

if (product.StockQuantity >= args.Quantity)

{

Console.WriteLine($"[Событие] Количество товара '{product.Name}' уменьшено на {args.Quantity}. Новый остаток: {product.StockQuantity}");

}

else

{

Console.WriteLine($"[Событие] Попытка уменьшить количество товара '{product.Name}' на {args.Quantity}, но недостаточно на складе (остаток: {product.StockQuantity})");

}

}

private static void ProductAddedToOrderHandler(object sender, ProductAddedToOrderEventArgs args)

{

var order = (Order)sender;

Console.WriteLine($"[Событие] Продукт '{args.Product.Name}' добавлен в заказ {order.OrderID}");

}

private static void OrderStatusUpdatedHandler(object sender, OrderStatusUpdatedEventArgs args)

{

var order = (Order)sender;

Console.WriteLine($"[Событие] Статус заказа {order.OrderID} изменен: {args.Old} -> {args.New}");

}

private static void CustomerAddedOrderHandler(object sender, CustomerAddedOrderEventArgs args)

{

var customer = (Customer)sender;

Console.WriteLine($"[Событие] Клиент {customer.Name} добавил новый заказ {args.Order.OrderID}");

}

}

**Шаг 7. Тестирование программы**

1. Запустите программу (**F5**).
2. Проверьте:
   * Уменьшается ли количество товара при добавлении в заказ.
   * Корректно ли отображаются данные клиента и заказа.
   * Правильно ли меняется статус заказа.

**Контрольные вопросы**

1. Что такое делегаты в C# и какова их основная цель?
2. Как объявить делегат и как его использовать?
3. В чем разница между многократным делегатом и делегатом с одним методом?
4. Что такое анонимные методы и как они отличаются от обычных методов?
5. Как создать анонимный метод и в каких случаях его следует использовать?
6. Что такое лямбда-выражения в C# и как они связаны с делегатами?
7. Какова синтаксическая структура лямбда-выражения?
8. В чем преимущества использования лямбда-выражений по сравнению с анонимными методами?