ФГАОУ ВО «КФУ им. В.И. Вернадского»
Физико-технический институт (структурное подразделение)

Кафедра компьютерной инженерии и моделирования

Скибинский Дмитрий Константинович

отчет по практической работе №6
по дисциплине **«ОБЪЕКТНО-ОРИЕНТИРОВАННОЕ ПРОГРАММИРОВАНИЕ»**

Направление подготовки:
09.03.04 "Программная инженерия"

Оценка - 100

Симферополь, 2024

Практическая работа №6
Тема: Делегаты, события, лямбда выражения, многопоточное и асинхронное программирование

Цель работы: Научиться на практике использованию делегатов, лямбда-выражений и событий. Научиться создавать собственные события.

Описание ключевых понятий:

Делегат — это тип, который представляет ссылку на методы с определенной сигнатурой. Он используется для передачи методов в качестве аргументов другим методам.

Мультикастовый делегат — это делегат, который может ссылаться на несколько методов одновременно. Он поддерживает операции добавления (+=) и удаления (-=) методов.

Callback — это метод, который передается другому методу и вызывается после завершения определенной операции.

+= — добавляет метод в цепочку делегатов.

-= — удаляет метод из цепочки делегатов.

Лямбда-выражение — это анонимная функция, которая может использоваться для создания делегатов или выражений запросов.

Событие — это механизм, который позволяет классу уведомлять другие классы о возникновении определенного действия.

EventHandler — это предопределенный делегат, который используется для обработки событий. Он принимает два параметра: объект-отправитель (sender) и объект EventArgs.

Sender — объект, который инициирует событие.

Receivers — объекты, которые подписываются на событие и обрабатывают его.

EventArgs — базовый класс для передачи данных в событиях. Его потомки (например, MouseEventArgs, KeyEventArgs) используются для передачи специфических данных.

Thread — это объект, представляющий поток выполнения. Потоки позволяют выполнять код параллельно.

lock — синхронизирует доступ к разделяемым ресурсам. **Join** — ожидает завершения потока.

Wait и **Pulse** — используются для синхронизации потоков с помощью объектов блокировки.

Mutex — примитив синхронизации, который может использоваться для защиты ресурсов в разных процессах. **Semaphore** — ограничивает количество потоков, которые могут одновременно обращаться к ресурсу.

TryEnter — пытается получить блокировку. **Priority** — устанавливает приоритет потока. **Background** — указывает, является ли поток фоновым.

Start — запускает поток. **IsAlive** — проверяет, выполняется ли поток.

Barrier — синхронизирует потоки, заставляя их ждать друг друга на определенном этапе.

TPL (Task Parallel Library) — библиотека для параллельного программирования.

Task — представляет асинхронную операцию.

Parallel — предоставляет методы для параллельного выполнения циклов.

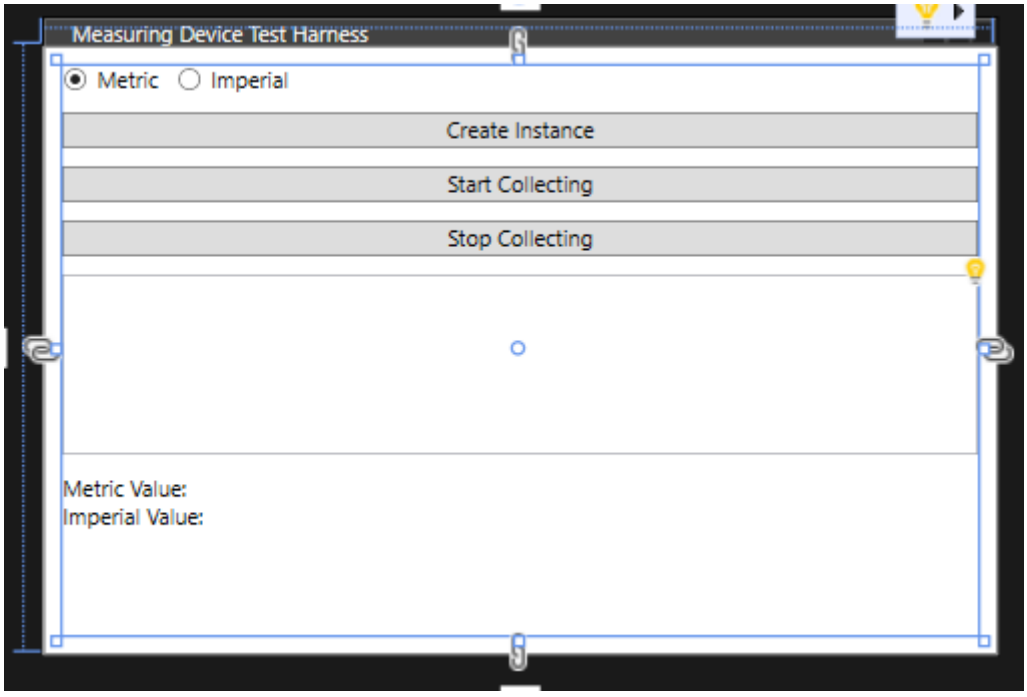
async и **await** — ключевые слова для асинхронного программирования.

Перед выполнением лабораторной работы изучена следующая литература:

1. Изучить презентации лектора курса: «Делегаты», «События», «Многопоточное программирование» (материалы доступны в "облаке" на Mail.ru и в Moodle КФУ).
2. Сайт Metanit.com
3. Справочник по C#. Корпорация Microsoft. http://msdn.microsoft.com/ru-ru/library/618ayhy6.aspx
4. Биллиг В.А. Основы программирования на C#. Интернет-университет информационных технологий. http://www.intuit.ru/studies/courses/2247/18/info
5. Павловская Т. Программирование на языке высокого уровня C#. http://www.intuit.ru/studies/courses/629/485/info
6. Руководство по программированию на C#. Корпорация Microsoft. http://msdn.microsoft.com/ru-ru/library/67ef8sbd.aspx
7. Корпорация Microsoft. C#. Спецификация языка. (Приложение A, Комментарии к документации).

Задание 1. Использование событий

Возьмем код программы из Лабораторной работы 5, с использованием интерфейса. Задача предполагает автоматический подсчет значений и уведомление клиента (пользовательского интерфейса) о новых измерениях. Поэтому мы можем убрать не нужные кнопки из пользовательского интерфейса для получения результата, так как он будет показываться автоматически с изменениями значений



Добавим интерфейс

```
namespace MeasuringDevice
{
    Ссылок: 2
    public interface IEventEnabledMeasuringDevice : IMeasuringDevice
    {
        event EventHandler NewMeasurementTaken;
    }
}
```

Новый интерфейс IEventEnabledMeasuringDevice является расширением существующего интерфейса IMeasuringDevice. Он добавляет поддержку событий, что позволяет классам, реализующим этот интерфейс, уведомлять другие части приложения о новых измерениях.

IEventEnabledMeasuringDevice наследует все методы и свойства из IMeasuringDevice. Событие NewMeasurementTaken типа EventHandler позволяет уведомлять подписанные методы о том, что устройство произвело новое измерение.

BackgroundWorker используется для асинхронного сбора данных от устройства. Сначала инициализируем его

```
private void GetMeasurements()
{
    dataCollector = new BackgroundWorker();
    dataCollector.WorkerSupportsCancellation = true;
    dataCollector.WorkerReportsProgress = true;
    dataCollector.DoWork += dataCollector_DoWork;
    dataCollector.ProgressChanged += dataCollector_ProgressChanged;
    dataCollector.RunWorkerAsync();
}
```

Добавим обработчик

```
private void dataCollector_DoWork(object sender, DoWorkEventArgs e)
{
    dataCaptured = new int[10];
    int i = 0;

    while (!dataCollector.CancellationPending && i < 10)
    {
        dataCaptured[i] = controller.TakeMeasurement();
        mostRecentMeasure = dataCaptured[i];

        dataCollector.ReportProgress(0);

        i++;

        // Задержка для уменьшения частоты измерений
        Thread.Sleep(1000); // 1 секунда
    }

    // Остановка сбора данных после 10 измерений
    if (i == 10)
    {
        dataCollector.CancelAsync();
    }
}
```

Сделаем инициализацию логирования

```
    // Инициализация StreamWriter для логирования
    if (!string.IsNullOrEmpty(LoggingFileName))
    {
        loggingFileWriter = new StreamWriter(LoggingFileName, true);
    }

    GetMeasurements();
}
```

Добавим логирование в методе

```
while (!dataCollector.CancellationPending && i < 10)
{
    dataCaptured[i] = controller.TakeMeasurement();
    mostRecentMeasure = dataCaptured[i];

    // Логирование измерения
    if (loggingFileWriter != null)
    {
        loggingFileWriter.WriteLine($"Measurement - {mostRecentMeasure}");
    }

    dataCollector.ReportProgress(0);

    i++;
```

Добавим закрытие файла лога

```
Ссылок: 2
public void StopCollecting()
{
    if (dataCollector != null)
    {
        dataCollector.CancelAsync();
    }

    // Закрытие файла лога
    if (loggingFileWriter != null)
    {
        loggingFileWriter.Close();
        loggingFileWriter = null;
    }
}
```

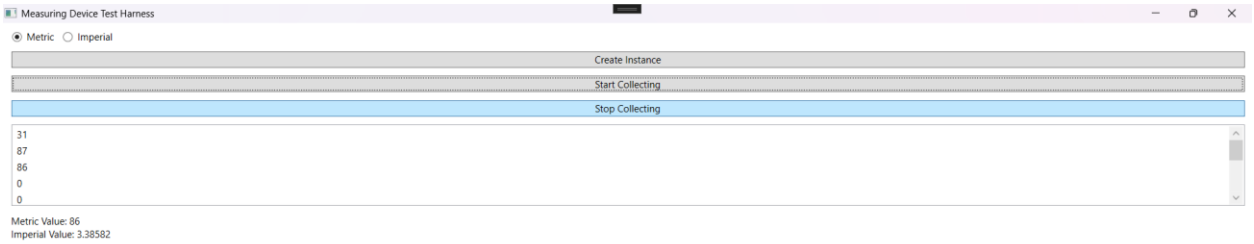
Освободим ресурсы

```
public void Dispose()
{
    if (dataCollector != null)
    {
        dataCollector.Dispose();
    }

    // Освобождение ресурсов StreamWriter
    if (loggingFileWriter != null)
    {
        loggingFileWriter.Close();
        loggingFileWriter.Dispose();
        loggingFileWriter = null;
    }
}
```

Сделай файл логирования

```
Ссылок: 1
public MeasureDataDevice(Units units)
{
    unitsToUse = units;
    LoggingFileName = "measurements.log";
}
```

Теперь наша программа автоматически подсчитывает значения Metric Value и Imperial Value



Задание 2. Использование лямбда-выражений

Задание 2 предполагает добавление поддержки события **HeartBeat** в приложение. Это событие будет генерироваться с определенным интервалом, чтобы уведомлять клиента о том, что устройство "живое". Будем использовать **лямбда-выражения** для обработки событий и **BackgroundWorker** для генерации пульса.

Этот класс будет использоваться для передачи данных о времени пульса.

```
using System;

namespace MeasuringDevice
{
    Ссылок: 3
    public class HeartBeatEventArgs : EventArgs
    {
        Ссылок: 2
        public DateTime TimeStamp { get; }

        Ссылок: 1
        public HeartBeatEventArgs()
        {
            TimeStamp = DateTime.Now;
        }
    }

    public delegate void HeartBeatEventHandler(object sender, HeartBeatEventArgs args);
}
```

Этот делегат будет использоваться для обработки события HeartBeat

В интерфейс добавим событие HeartBeat и свойство HeartBeatInterval в интерфейс

```
namespace MeasuringDevice
{
    Ссылок: 2
    public interface IEventEnabledMeasuringDevice : IMeasuringDevice
    {
        event EventHandler NewMeasurementTaken;
        event HeartBeatEventHandler HeartBeat;
        Ссылок: 3
        int HeartBeatInterval { get; }
    }
}
```

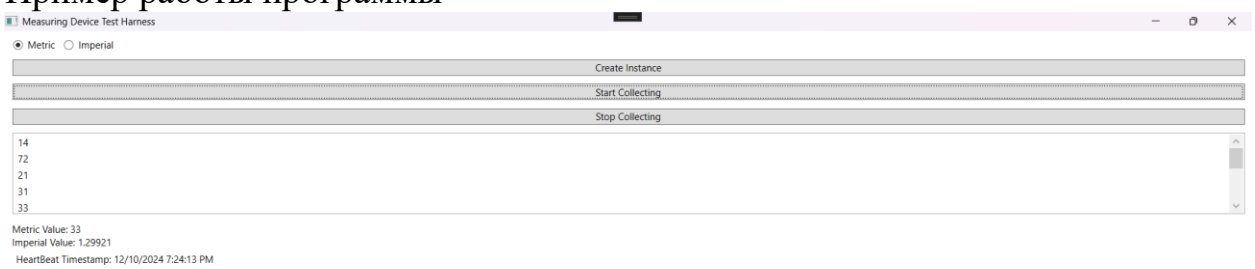
Добавим **HeartBeat** и **HeartBeatInterval** в класс **MeasureDataDevice**

```
Ссылок: 3
public abstract class MeasureDataDevice : IEventEnabledMeasuringDevice, IDisposable
{
    private Units unitsToUse;
    private int[] dataCaptured;
    private int mostRecentMeasure;
    private DeviceController.DeviceController controller;
    private const DeviceType measurementType = DeviceType.LENGTH;
    private BackgroundWorker dataCollector;
    private BackgroundWorker heartBeatTimer;
    private int heartBeatIntervalTime;

    public event EventHandler NewMeasurementTaken = delegate { };
    public event HeartBeatEventHandler HeartBeat = delegate { };

    Ссылок: 3
    public int HeartBeatInterval
    {
        get { return heartBeatIntervalTime; }
        protected set { heartBeatIntervalTime = value; }
    }
```

Конструктор класса MeasureLengthDevice отвечает за инициализацию объекта класса. Конструктор принимает параметр heartBeatInterval, который определяет интервал времени (в миллисекундах) между генерацией события HeartBeat. Это значение передается в базовый класс через свойство HeartBeatInterval.

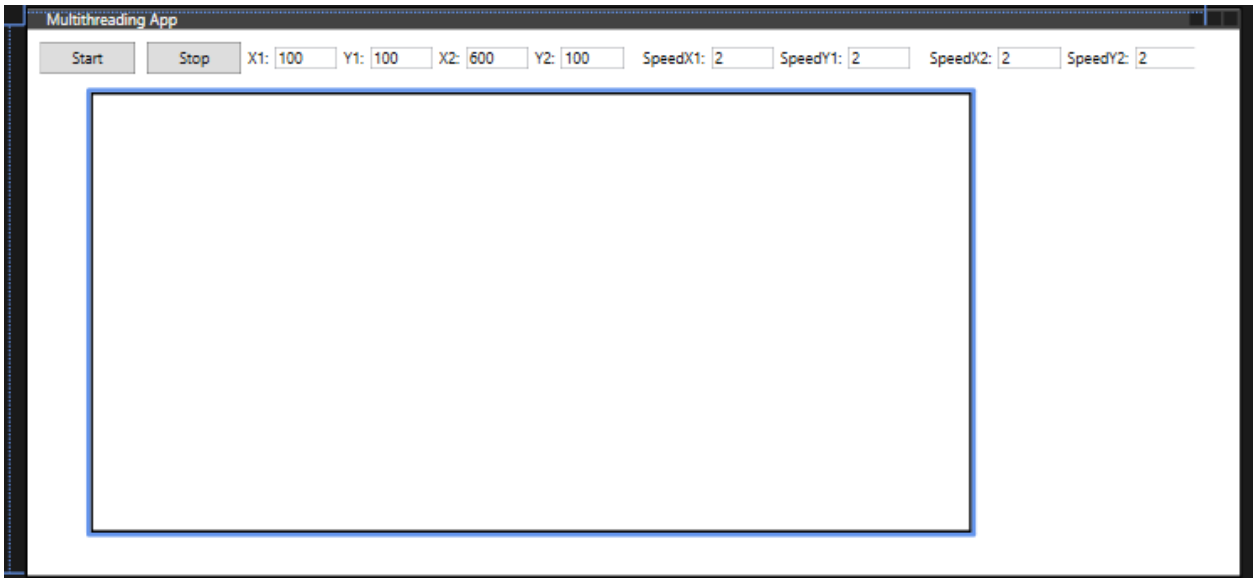
```
using DeviceController;
using System;
using System.Threading;

namespace MeasuringDevice
{
    Ссылок: 5
    public class MeasureLengthDevice : MeasureDataDevice
    {
        Ссылок: 1
        public MeasureLengthDevice(Units units, int heartBeatInterval) : base(units)
        {
            HeartBeatInterval = heartBeatInterval;
        }

        Ссылок: 2
        public MeasureLengthDevice(Units units) : this(units, 1000)
        {
        }
    }
}
```

Пример работы программы



Задание 3. Создание многопоточных приложений и асинхронных методов.

Сделаем приложение, которое моделирует движение двух шариков (Ball) на холсте (Canvas). Шарики могут перемещаться, отскакивать от границ холста и сталкиваться друг с другом.



Метод StartButton_Click

Этот метод обрабатывает нажатие кнопки "Start". Он инициализирует шарики, добавляет их на холст и запускает их движение. Происходит валидация введенных значений скорости. Инициализация шариков и добавления их на холст. Запуск движения шаров. Используется асинхронный запуск с помощью Task.Run. Движение шариков выполняется параллельно с использованием Parallel.For.

```
private async void StartButton_Click(object sender, RoutedEventArgs e)
{
    if (!double.TryParse(X1TextBox.Text, out double x1) || !double.TryParse(Y1TextBox.Text, out double y1) ||
        !double.TryParse(X2TextBox.Text, out double x2) || !double.TryParse(Y2TextBox.Text, out double y2) ||
        !double.TryParse(SpeedX1TextBox.Text, out double speedX1) || !double.TryParse(SpeedY1TextBox.Text, out double speedY1) ||
        !double.TryParse(SpeedX2TextBox.Text, out double speedX2) || !double.TryParse(SpeedY2TextBox.Text, out double speedY2))
    {
        MessageBox.Show("Please enter valid numeric coordinates and speeds.");
        return;
    }

    isRunning = true;
    var ball1 = new Ball(x1, y1, speedX1, speedY1, Brushes.Red);
    var ball2 = new Ball(x2, y2, speedX2, speedY2, Brushes.Blue);
    balls.Clear();
    balls.Add(ball1);
    balls.Add(ball2);

    foreach (var ball in balls)
    {
        var ellipse = new Ellipse
        {
            Width = 20,
            Height = 20,
            Fill = ball.Color
        };
        Canvas.SetLeft(ellipse, ball.X);
        Canvas.SetTop(ellipse, ball.Y);
        ellipse.MouseLeftButtonDown += Ellipse_MouseLeftButtonDown;
        ellipse.MouseMove += Ellipse_MouseMove;
        ellipse.MouseLeftButtonUp += Ellipse_MouseLeftButtonUp;
        canvas.Children.Add(ellipse);
    }

    // Use Parallel.For to process each ball concurrently
    await Task.Run(() =>
    {
        Parallel.For(0, balls.Count, i =>
        {
            MoveBall(balls[i]);
        });
    });
}
```

Метод MoveBall(Ball ball)

Этот метод отвечает за перемещение шарика и обновление его позиции на холсте.

Цикл движения: Метод работает в бесконечном цикле, пока флаг isRunning равен true. Внутри цикла вызывается метод ball.Move(), который обновляет координаты шарика.

Обновление позиции на холсте: Для обновления позиции шарика на холсте используется Dispatcher.Invoke. Это необходимо, так как обновление UI должно выполняться в основном потоке.

Задержка: Используется Thread.Sleep(30) для замедления движения шариков. Это заменяет Task.Delay, так как Parallel.For работает с потоками, а не с задачами.

```
Ссылок 1
private void MoveBall(Ball ball)
{
    while (isRunning)
    {
        ball.Move(balls, mutex); // Pass the mutex for synchronization
        Dispatcher.Invoke(() => UpdateBallPosition(ball));
        Thread.Sleep(30); // Replaced Task.Delay with Thread.Sleep for Parallel.For
    }
}
```

Метод UpdateBallPosition(Ball ball)
Этот метод обновляет позицию шарика на холсте.
Поиск элемента Ellipse: Метод находит элемент Ellipse, соответствующий шарику, по цвету.
Обновление позиции: Если элемент найден, его позиция на холсте обновляется с использованием методов Canvas.SetLeft и Canvas.SetTop.

```
Ссылок: 1
private void UpdateBallPosition(Ball ball)
{
    var ellipse = canvas.Children.OfType<Ellipse>().FirstOrDefault(el => el.Fill == ball.Color);
    if (ellipse != null)
    {
        Canvas.SetLeft(ellipse, ball.X);
        Canvas.SetTop(ellipse, ball.Y);
    }
}
```

По поводу физической составляющей программы. Используется отражения от сторон поля

```
public void Move(List<Ball> balls, Mutex mutex)
{
    mutex.WaitOne(); // Wait for the mutex

    X += SpeedX;
    Y += SpeedY;

    // Отражение от границ квадрата
    if (X < 0 || X > 680) SpeedX = -SpeedX;
    if (Y < 2 || Y > 330) SpeedY = -SpeedY;

    // Отталкивание от других шариков
    foreach (var otherBall in balls)
```

По закону

$$v'_x = -v_x$$
$$v'_y = -v_y$$

- **Где:**
  - v_x — горизонтальная скорость шарика до столкновения.
  - v_y — вертикальная скорость шарика до столкновения.
  - v'_x — горизонтальная скорость шарика после столкновения.
  - v'_y — вертикальная скорость шарика после столкновения.

Измеряем расстояние между шарами

Формула:

Для упругого столкновения двух шариков используются формулы для сохранения импульса и энергии:

$$u_{1x} = \frac{v_{1x} \cdot (m_1 - m_2) + 2 \cdot m_2 \cdot v_{2x}}{m_1 + m_2}$$
$$u_{2x} = \frac{v_{2x} \cdot (m_2 - m_1) + 2 \cdot m_1 \cdot v_{1x}}{m_1 + m_2}$$

- **Где:**
  - u_{1x}, u_{2x} — проекции новых скоростей шариков на ось X.
  - v_{1x}, v_{2x} — проекции скоростей шариков на ось X до столкновения.
  - m_1, m_2 — массы шариков.

Расчитываним сталкивание шаров

```
double angle = Math.Atan2(dy, dx);
double overlap = Radius * 2 - distance;

// Перемещаем шарики, чтобы они не пересекались
X -= overlap * Math.Cos(angle) / 2;
Y -= overlap * Math.Sin(angle) / 2;
otherBall.X += overlap * Math.Cos(angle) / 2;
otherBall.Y += overlap * Math.Sin(angle) / 2;
```

Формула:

Шарики перемещаются вдоль линии, соединяющей их центры, чтобы избежать перекрытия:

$$\Delta x = \frac{(2r - d) \cdot \cos(\theta)}{2}$$
$$\Delta y = \frac{(2r - d) \cdot \sin(\theta)}{2}$$

- **Где:**
  - $\Delta x, \Delta y$ — смещение шариков по осям X и Y.
  - θ — угол между линией, соединяющей центры шариков, и осью X.
  - $2r - d$ — величина перекрытия шариков.

Вычисляем новые скорости шаров после столкновения

```
// Вычисляем новые скорости после столкновения
double v1 = Math.Sqrt(SpeedX * SpeedX + SpeedY * SpeedY);
double v2 = Math.Sqrt(otherBall.SpeedX * otherBall.SpeedX + otherBall.SpeedY * otherBall.SpeedY);
double theta1 = Math.Atan2(SpeedY, SpeedX);
double theta2 = Math.Atan2(otherBall.SpeedY, otherBall.SpeedX);
double phi = Math.Atan2(dy, dx);
```

Формула:

Для упругого столкновения двух шариков используются формулы для сохранения импульса и энергии:

$$u_{1x} = \frac{v_{1x} \cdot (m_1 - m_2) + 2 \cdot m_2 \cdot v_{2x}}{m_1 + m_2}$$
$$u_{2x} = \frac{v_{2x} \cdot (m_2 - m_1) + 2 \cdot m_1 \cdot v_{1x}}{m_1 + m_2}$$

- **Где:**
  - u_{1x}, u_{2x} — проекции новых скоростей шариков на ось X.
  - v_{1x}, v_{2x} — проекции скоростей шариков на ось X до столкновения.
  - m_1, m_2 — массы шариков.

Обновление скоростей

```
double v1x = v1 * Math.Cos(theta1 - phi);
double v1y = v1 * Math.Sin(theta1 - phi);
double v2x = v2 * Math.Cos(theta2 - phi);
double v2y = v2 * Math.Sin(theta2 - phi);

double u1x = (v1x * (Mass - otherBall.Mass) + 2 * otherBall.Mass * v2x) / (Mass + otherBall.Mass);
double u2x = (v2x * (otherBall.Mass - Mass) + 2 * Mass * v1x) / (Mass + otherBall.Mass);

SpeedX = u1x * Math.Cos(phi) + v1y * Math.Cos(phi + Math.PI / 2);
SpeedY = u1x * Math.Sin(phi) + v1y * Math.Sin(phi + Math.PI / 2);
otherBall.SpeedX = u2x * Math.Cos(phi) + v2y * Math.Cos(phi + Math.PI / 2);
otherBall.SpeedY = u2x * Math.Sin(phi) + v2y * Math.Sin(phi + Math.PI / 2);
```

Формула:

Новые скорости шариков вычисляются с учетом угла столкновения:

$$v'_x = u_x \cdot \cos(\phi) + v_y \cdot \cos\left(\phi + \frac{\pi}{2}\right)$$
$$v'_y = u_x \cdot \sin(\phi) + v_y \cdot \sin\left(\phi + \frac{\pi}{2}\right)$$

- **Где:**
  - v'_x, v'_y — новые скорости шариков.
  - u_x — проекция новой скорости на ось X.
  - v_y — вертикальная скорость шарика до столкновения.
  - ϕ — угол между линией, соединяющей центры шариков, и осью X.

Мьютекс используется в методе MoveBall для синхронизации доступа к списку шариков (balls).
Если не использовать мьютекс, то два шарика могут одновременно пытаться изменить свои координаты или взаимодействовать друг с другом, что приведет к некорректному поведению. Например:

- Один шарик может перекрыть другой, неправильно вычислив расстояние между ними.
- Координаты шариков могут быть повреждены из-за одновременного доступа.

В методе StartButton_Click вы запускаете движение шариков асинхронно с помощью Task.Run

Если выполнять движение шариков в основном потоке, то приложение "зависнет", пока шарики двигаются. Использование Task.Run позволяет выполнять движение в фоновом потоке.

Параллельное выполнение: Вы используете Parallel.For, чтобы запустить движение каждого шарика в отдельном потоке. Это ускоряет выполнение, так как шарики двигаются независимо друг от друга.

async и await для запуска движения шариков асинхронно

Ответы на вопросы:

1. Что такое delegate?

Delegate — это тип, который представляет ссылку на методы с определенной сигнатурой. Он используется для передачи методов в качестве аргументов другим методам.

2. Что такое функции обратного вызова (callback)?

Функция обратного вызова (callback) — это функция, которая передается другому методу и вызывается после завершения определенной операции.

3. Как строятся цепочки делегатов?

Цепочки делегатов создаются с помощью операторов += (добавление метода в цепочку) и -= (удаление метода из цепочки).

4. Что такое анонимная функция? Приведите пример.

Анонимная функция — это функция, которая не имеет имени и объявляется "на лету". В C# анонимные функции могут быть реализованы с помощью лямбда-выражений или анонимных методов.

```
Action<string> anonymousFunction = (message) => Console.WriteLine(message);
anonymousFunction("Hello from anonymous function!");
```

5. Напишите пример лямбда-выражения.

Лямбда-выражение — это компактный способ записи анонимной функции.

Пример:

```
Func<int, int, int> add = (x, y) => x + y;
Console.WriteLine(add(5, 3)); // Вывод: 8
```

6. Связывание обработчика с событием.

Обработчик события связывается с событием с помощью оператора +=.

7. Отключение обработчика. Динамическое связывание событий с их обработчиками.

Обработчик отключается с помощью оператора -=. Динамическое связывание и отключение обработчиков позволяет управлять событиями во время выполнения программы.

8. Как реализуется блокировка потока?

Блокировка потока реализуется с помощью ключевого слова lock или объектов синхронизации, таких как Mutex и Semaphore.

9. Что такое мьютекс и семафор?

Мьютекс (Mutex):

1)Используется для синхронизации доступа к общим ресурсам.
2)Позволяет только одному потоку входить в критическую секцию.

Семафор (Semaphore):

Позволяет ограниченному числу потоков входить в критическую секцию.

10. В чем преимущества использования новой парадигмы многопоточности в C# с использованием библиотеки TPL?

TPL (Task Parallel Library) — это библиотека для параллельного программирования в C#. Ее преимущества:

1.Упрощение многопоточности: TPL автоматически управляет потоками, что упрощает написание параллельного кода.

2. Высокая производительность: TPL оптимизирует использование потоков, что повышает производительность.

3. Поддержка асинхронности: TPL поддерживает асинхронное программирование с использованием Task и async/await.

11. Что такое Parallel.For?

Parallel.For — это метод из TPL, который позволяет выполнять цикл параллельно.

12. Что означают термины async и await?

async — ключевое слово, которое указывает, что метод является асинхронным.
await — оператор, который приостанавливает выполнение метода до завершения асинхронной операции.

Представлены 4 проекта, реализованных в Visual Studio Community 2022. Проекты представлены преподавателю в электронной форме, продемонстрирована их работоспособность, разъяснены детали программного кода.