Тема 7. Таймеры. Асинхронность в Windows Forms.

Цель занятия:

Ознакомиться с основами работы с таймерами и асинхронным программированием в Windows Forms.

Учебные вопросы:

1. Использование Timer для создания событий через интервалы времени.

2. Основные понятия асинхронного программирования.

3. Асинхронные операции в Windows Forms.

4. Асинхронные возможности C#.

5. Работа с длительными операциями

6. Примеры из практики

1. Использование Timer для создания событий через интервалы времени.

Класс Timer в Windows Forms используется для выполнения операций через заданные интервалы времени.

Этот класс удобен, например, для создания анимаций, работы с часами или выполнения фоновых задач.

Основные свойства и методы Timer:

- **Interval** — задает интервал времени (в миллисекундах), через который будет вызываться событие Tick. (Например, Interval = 1000; — вызов события каждую секунду.)
- **Enabled** — свойство, которое включает (true) или выключает (false) таймер.
- **Start()** — метод для запуска таймера.
- **Stop()** — метод для остановки таймера.
- Событие **Tick** — возникает каждый раз, когда истекает интервал, заданный в свойстве Interval.

Простой пример:

```
using Timer = System.Windows.Forms.Timer;
namespace MyTimer
{
    Ссылок: 3
    public partial class Form1 : Form
    {
        //поле timer в классе, которое будет использоваться
        //для работы с объектом типа Timer
        private Timer timer;
        Ссылок: 1
        public Form1()
        {
            InitializeComponent();

            // Инициализация Timer
            timer = new Timer();
            timer.Interval = 1000; // Интервал 1 секунда
            timer.Tick += Timer_Tick; // Подписка на событие Tick
            timer.Start(); // Запуск таймера
        }

        Ссылок: 1
        private void Timer_Tick(object sender, EventArgs e)
        {
            label1.Text = "Таймер сработал!";
        }
    }
}
```

Анимация перемещения кнопки по форме:

```
// создание псевдонима (короткого имени) (Timer)
// для полного имени класса (System.Windows.Forms.Timer).
using Timer = System.Windows.Forms.Timer;

namespace lesson8timers
{
    Ссылок: 3
    public partial class Form1 : Form
    {
        private Timer timer; // объявление приватного поля timer в классе.
        private int direction = 1; // Направление движения (1 - вправо,
                                   // -1 - влево

        Ссылок: 1
        public Form1()
        {
```

```
public Form1()
{
    InitializeComponent();

    // Настройка таймера
    timer = new Timer // создаётся новый экземпляр класса Timer
    {
        Interval = 10 // Интервал в миллисекундах (10 мс)
    };
    timer.Tick += Timer_Tick; // Подписка на событие Tick
    timer.Start(); // Запуск таймера
}
```

```
private void Timer_Tick(object sender, EventArgs e)
{
    // Перемещение кнопки
    int newX = button1.Location.X + direction * 2; // Изменение координаты X
    button1.Location = new Point(newX, button1.Location.Y);

    // Проверка границ формы
    if (newX <= 0 || newX + button1.Width >= this.ClientSize.Width)
    {
        direction *= -1; // Смена направления движения
    }
}
```

2. Основные понятия асинхронного программирования.

Асинхронное программирование — это подход, позволяющий выполнять задачи без блокировки основного потока выполнения программы. Оно позволяет:

- Не блокировать основной поток (UI-поток): Приложение продолжает реагировать на действия пользователя, пока выполняются длительные операции.
- Оптимально использовать ресурсы: Асинхронность позволяет избежать простаивания ресурсов, например, ожидания ответа от сети или завершения ввода-вывода.
- Повышать производительность: Несколько операций могут выполняться параллельно, что ускоряет обработку данных.

Зачем нужна асинхронность в Windows Forms

Windows Forms работает на единственном UI-потоке, который отвечает за:

- Отрисовку пользовательского интерфейса.
- Обработку событий (нажатий кнопок, перемещения мыши и т.д.).

Если в этом потоке выполняется длительная операция (например, чтение файла, запрос к базе данных или загрузка данных из интернета), то:

- UI "замораживается", и приложение перестает реагировать на действия пользователя.
- Появляется надпись "Не отвечает" в заголовке окна.
- Пользовательский опыт ухудшается.

Асинхронность решает эти проблемы:

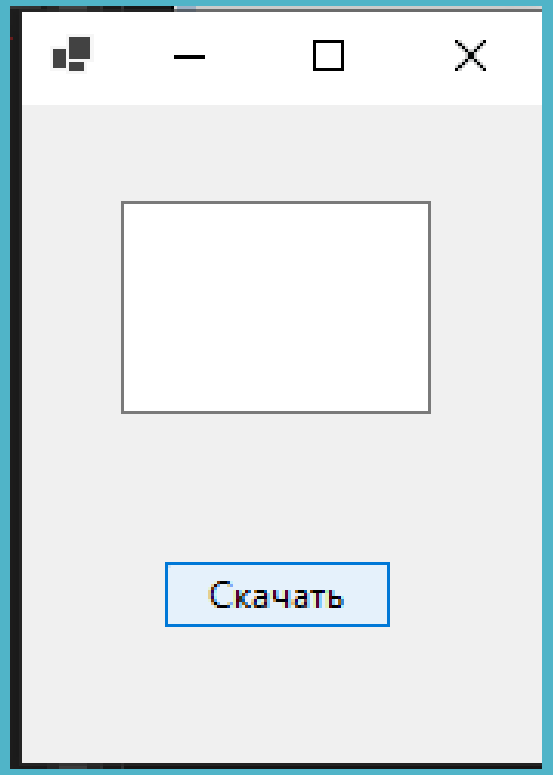
- Длительные задачи выполняются в фоне, не мешая UI.
- UI остается отзывчивым.
- Повышается производительность и удобство использования приложения.

Проблемы блокировки пользовательского интерфейса (UI):

- Длительные операции в основном потоке. Пример: Вызов метода для скачивания файла в UI-потоке вызывает блокировку интерфейса до завершения операции. Пользователь видит замороженное окно, которое перестает реагировать.
- Проблемы многопоточности. Доступ к элементам UI возможен только из основного (UI) потока. Если другой поток пытается изменить элемент интерфейса, это вызывает исключение InvalidOperationException.
- Ухудшение UX (пользовательского опыта). Пользователь не получает визуальной обратной связи о том, что операция выполняется (например, ProgressBar не обновляется). Пользователь может решить закрыть приложение, думая, что оно "зависло".

Пример синхронного кода. После нажатия кнопки, до окончания длительной операции, интерфейс будет заблокирован, окно textBox не будет реагировать на ввод.

```
Ссылок: 1
void button1_Click(object sender, EventArgs e)
{
    // Прямая блокировка основного потока
    Thread.Sleep(5000); // Симуляция долгой операции
    MessageBox.Show("Операция завершена!");
}
```



Если обработчик события сделать асинхронным, то в этом случае UI остается отзывчивым, так как операция выполняется асинхронно.

```
Ссылок: 1
async void button1_Click(object sender, EventArgs e)
{
    await Task.Delay(5000); // Асинхронное ожидание
    MessageBox.Show("Операция завершена!");
}
```

3. Асинхронные операции в Windows Forms

Существует несколько подходов к реализации асинхронных операций: использование событий, таймеров, потоков и задач. Рассмотрим основные из них.

1. Использование событий и таймеров.

Подходит для простых периодических задач, которые требуют выполнения через определенные интервалы времени.

Примеры: обновление интерфейса (часы, счетчики), регулярные проверки состояния.

Основной инструмент: System.Windows.Forms.Timer.

Плюсы:

- Простота.
- Полностью интегрирован с UI-потоком.

Минусы:

- Неподходит для длительных или ресурсоемких операций.
- Не гибок для сложных задач.

Пример:

```
private Timer timer;

private void Form1_Load(object sender, EventArgs e)
{
    timer = new Timer();
    timer.Interval = 1000; // 1 секунда
    timer.Tick += Timer_Tick;
    timer.Start();
}

private void Timer_Tick(object sender, EventArgs e)
{
    labelTime.Text = DateTime.Now.ToString("HH:mm:ss");
}
```

2. Использование потоков (Threads).

Дает полный контроль над фоновыми операциями.

Подходит для длительных операций, таких как чтение файлов, обработка данных и запросы к базе данных.

Основная проблема: необходимость взаимодействия с UI через Invoke.

Плюсы:

- Подходит для задач, требующих выполнения на фоне.
- Полный контроль над процессом.

Минусы:

- Сложности в работе с UI.
- Ручное управление потоками.

Пример:

```
private void StartButton_Click(object sender, EventArgs e)
{
    Thread thread = new Thread(DoWork);
    thread.Start();
}

private void DoWork()
{
    Thread.Sleep(3000); // Длительная операция
    UpdateLabel("Готово");
}

private void UpdateLabel(string text)
{
    if (label.InvokeRequired)
    {
        label.Invoke(new Action(() => label.Text = text));
    }
    else
    {
        label.Text = text;
    }
}
```

3. Использование BackgroundWorker.

Упрощенный инструмент для выполнения задач в фоновом режиме.

Автоматически предоставляет методы для безопасного взаимодействия с UI.

События:

- DoWork — выполнение фоновой задачи.
- ProgressChanged — обновление прогресса.
- RunWorkerCompleted — завершение задачи.

Плюсы:

- Простота и безопасность.
- Встроенная поддержка событий.

Минусы:

- Ограниченная гибкость.
- Устаревший подход (современные приложения предпочитают Task и async/await).

Пример:

```
private void StartButton_Click(object sender, EventArgs e)
{
    BackgroundWorker worker = new BackgroundWorker();
    worker.DoWork += Worker_DoWork;
    worker.RunWorkerCompleted += Worker_RunWorkerCompleted;
    worker.RunWorkerAsync();
}

private void Worker_DoWork(object sender, DoWorkEventArgs e)
{
    Thread.Sleep(3000); // Длительная операция
    e.Result = "Готово";
}

private void Worker_RunWorkerCompleted(object sender, RunWorkerCompletedEventArgs e)
{
    label.Text = e.Result.ToString(); // Безопасное обновление UI
}
```

4. Использование Task и async/await.

Современный подход к асинхронности.

Обеспечивает простую запись асинхронного кода.

Автоматически возвращает выполнение в UI-поток после завершения асинхронной операции.

Плюсы:

- Читаемый и понятный код.
- Безопасное обновление UI без явного использования Invoke.
- Простое управление сложными задачами.

Минусы:

- Требует понимания работы с асинхронностью и Task.

Пример:

```
Ссылок: 0
async void StartButton_Click(object sender, EventArgs e)
{
    label1.Text = "Выполняется...";
    await Task.Delay(3000); // Асинхронная операция
    label1.Text = "Готово!";
}
```

Основные термины: Task, async/await

Task — это класс в .NET, представляющий асинхронную операцию. Он используется для выполнения задач, которые могут выполняться параллельно или асинхронно, без блокировки основного потока.

Основные характеристики:

- Task может возвращать результат (если используется Task<TResult>).
- Поддерживает отмену с помощью CancellationToken.
- Позволяет отслеживать состояние задачи (ожидание, выполнение, завершение).

Пример:

```
Task<int> task = Task.Run(() => CalculateSomething());
int result = await task; // Ожидание завершения задачи и получение результата.
```

Task.Run запускает указанный код (CalculateSomething()) в отдельном потоке. Это позволяет выполнять длительные или ресурсоемкие операции без блокировки основного потока.

Task.Run возвращает объект типа Task<int>, который представляет асинхронную операцию, которая в конечном итоге возвращает результат типа int.

async

Ключевое слово **async** используется для обозначения асинхронного метода. Оно указывает компилятору, что метод содержит асинхронные операции и может использовать **await**.

Особенности:

- Метод, помеченный async, должен возвращать Task, Task<T> или void (последнее не рекомендуется).
- async не делает метод асинхронным сам по себе, он лишь позволяет использовать await

await

Ключевое слово **await** используется внутри асинхронного метода для приостановки его выполнения до завершения асинхронной операции. При этом управление возвращается вызывающему коду, что позволяет избежать блокировки потока.

Особенности:

- **await** можно использовать только внутри методов, помеченных **async**.
- После завершения асинхронной операции выполнение метода возобновляется.
- Если асинхронная операция возвращает результат, **await** извлекает его.

Связь между Task, async и await

- **Task** представляет асинхронную операцию.
- **async** указывает, что метод может содержать асинхронные операции.
- **await** приостанавливает выполнение метода до завершения Task.

Ключевое слово async указывает компилятору, что метод является асинхронным.

```
async void getButton_Click(object sender, RoutedEventArgs e)
{
    var w = new WebClient();

    string txt = await w.DownloadStringTaskAsync("...");

    dataTextBox.Text = txt;
}
```

await указывает компилятору, что в этой точке необходимо дождаться окончания асинхронной операции (при этом управление возвращается вызвавшему методу).

```
async void DoDownloadAsync()
{
    using (var w = new WebClient())
    {
        string txt = await w.DownloadStringTaskAsync("http://www.microsoft.com/");
        dataTextBox.Text = txt;
    }
}

void DoDownload()
{
    using (var w = new WebClient())
    {
        string txt = w.DownloadString("http://www.microsoft.com/");
        dataTextBox.Text = txt;
    }
}
```

Еще примеры:

```
Ссылок: 0
public async Task<string> DownloadDataAsync(string url)
{
    using (HttpClient client = new HttpClient())
    {
        // Ожидание завершения HTTP-запроса.
        string data = await client.GetStringAsync(url);
        return data;
    }
}
```

Когда вызывается метод DownloadDataAsync, он создает объект HttpClient и выполняет асинхронный HTTP-запрос по указанному URL.

Достигнув строки await client.GetStringAsync(url);, метод приостанавливает свое выполнение и возвращает управление вызывающему коду.

После завершения HTTP-запроса выполнение метода продолжается, и результат (строка data) возвращается.

```
Ссылок: 0
public async Task DoSomethingAsync()
{
    await Task.Delay(1000); // Асинхронная задержка.
}
```

Когда вызывается метод DoSomethingAsync, он начинает выполняться.

Достигнув строки await Task.Delay(1000);, метод приостанавливает свое выполнение и возвращает управление вызывающему коду.

Через 1 секунду задача Task.Delay(1000) завершается, и выполнение метода DoSomethingAsync продолжается.

Поскольку метод не содержит кода после await, он завершается сразу после завершения задержки.

5. Использование потокобезопасных библиотек.

Для сложных многопоточных приложений могут использоваться потокобезопасные коллекции (ConcurrentQueue, BlockingCollection) и библиотеки для управления потоками.

Примеры: при создании очередей задач или управлении большим числом операций.

Плюсы:

- Подходит для сложных сценариев.
- Потокобезопасность.

Минусы:

- Необходимость сложной архитектуры.

Рекомендации:

- Для простых задач: используйте таймеры или BackgroundWorker.
- Для современных приложений: предпочтителен async/await с Task.
- Для сложных многопоточных задач: используйте потоки или специальные библиотеки.

4. Асинхронные возможности C#.

Ключевые слова async и await

async:

- Указывает, что метод является асинхронным. Это позволяет использовать ключевое слово await внутри метода.
- Метод, помеченный async, должен возвращать Task, Task<T> или void (последнее не рекомендуется).

await:

- Приостанавливает выполнение асинхронного метода до завершения задачи (Task). При этом управление возвращается вызывающему коду, что позволяет избежать блокировки потока.
- После завершения задачи выполнение метода продолжается.

Пример:

```
public async Task DoSomethingAsync()
{
    await Task.Delay(1000); // Асинхронная задержка на 1 секунду.
}
```

Типы задач: Task и Task<T>

Task:

- Представляет асинхронную операцию, которая не возвращает результат.
- Используется для выполнения операций, которые нужно выполнить асинхронно, но результат которых не требуется.

Task<T>:

- Представляет асинхронную операцию, которая возвращает результат типа T.
- Используется для выполнения операций, которые возвращают данные (например, загрузка данных из сети).

Пример Task:

```
public async Task DoWorkAsync()
{
    await Task.Run(() => Console.WriteLine("Работа выполнена!"));
}
```

Пример Task<T>:

```
public async Task<int> CalculateAsync()
{
    return await Task.Run(() => 42); // Возвращает число 42.
}
```

Примеры использования асинхронных методов в Windows Forms

Пример 1: **Асинхронная загрузка данных из сети**

Задача: Загрузить данные с веб-сайта и отобразить их в текстовом поле без блокировки UI-потока.

```
public async Task<string> DownloadDataAsync(string url)
{
    using (HttpClient client = new HttpClient())
    {
        string data = await client.GetStringAsync(url); // Асинхронный HTTP-запрос.
        return data;
    }
}

private async void button1_Click(object sender, EventArgs e)
{
    string url = "https://example.com/data";
    string data = await DownloadDataAsync(url); // Ожидание завершения загрузки.
    textBox1.Text = data; // Отображение данных в текстовом поле.
}
```

Пример 2: **Асинхронная обработка файлов**

Задача: Прочитать содержимое файла асинхронно и отобразить его в текстовом поле.

```
public async Task<string> ReadFileAsync(string filePath)
{
    using (StreamReader reader = new StreamReader(filePath))
    {
        string content = await reader.ReadToEndAsync(); // Асинхронное чтение файла.
        return content;
    }
}

private async void button1_Click(object sender, EventArgs e)
{
    string filePath = "example.txt";
    string content = await ReadFileAsync(filePath); // Ожидание завершения чтения.
    textBox1.Text = content; // Отображение содержимого файла.
}
```

Пример 3: **Асинхронная задержка с обновлением UI**

Задача: Выполнить задержку в 5 секунд с обновлением прогресса на ProgressBar.

```
public async Task DelayWithProgressAsync(IProgress<int> progress)
{
    for (int i = 0; i <= 100; i++)
    {
        await Task.Delay(50); // Асинхронная задержка.
        progress.Report(i); // Обновление прогресса.
    }
}

private async void button1_Click(object sender, EventArgs e)
{
    var progress = new Progress<int>(value => progressBar1.Value = value);
    await DelayWithProgressAsync(progress); // Ожидание завершения задержки.
    MessageBox.Show("Задержка завершена!");
}
```

Итог

- Ключевые слова async и await позволяют легко писать асинхронный код.
- Типы Task и Task<T> представляют асинхронные операции.
- В Windows Forms асинхронные методы используются для выполнения длительных операций без блокировки UI-потока, что делает приложение более отзывчивым и удобным для пользователя.

5. Работа с длительными операциями

Пример 1: Асинхронная загрузка данных из сети

Создание асинхронного метода для загрузки данных:

```
public async Task<string> DownloadDataAsync(string url)
{
    using (HttpClient client = new HttpClient())
    {
        string data = await client.GetStringAsync(url);
        return data;
    }
}
```

Вызов асинхронного метода из обработчика событий кнопки:

```
private async void buttonLoadData_Click(object sender, EventArgs e)
{
    string url = "https://example.com/data";
    string data = await DownloadDataAsync(url);
    UpdateUIWithDownloadedData(data);
}
```

Обновление UI после завершения операции:

```
private void UpdateUIWithDownloadedData(string data)
{
    textBoxData.Text = data;
}
```

Пример 2: Обработка больших объемов данных в асинхронном режиме

Создание асинхронного метода для обработки данных:

```
public async Task<int> ProcessDataAsync(List<int> data)
{
    return await Task.Run(() =>
    {
        // Длительная операция обработки данных
        int result = 0;
        foreach (int value in data)
        {
            result += value;
        }
        return result;
    });
}
```

Вызов асинхронного метода из обработчика событий кнопки:

```
private async void buttonProcessData_Click(object sender, EventArgs e)
{
    List<int> data = GetDataToProcess();
    int result = await ProcessDataAsync(data);
    UpdateUIWithProcessingResult(result);
}
```

Обновление UI после завершения операции:

```
private void UpdateUIWithProcessingResult(int result)
{
    labelResult.Text = $"Результат: {result}";
}
```

6. Примеры из практики

Пример 1: Асинхронная загрузка файла с сервера.

Пример 2: Асинхронный доступ к базе данных.

Пример 3: Уведомления о прогрессе выполнения длительных операций.

Список литературы:

1. MSDN: async
2. MSDN: await
3. Асинхронное программирование с использованием ключевых слов Async и Await
4. Асинхронные методы, async и await
5. Уроки C# – операторы async await
6. Видеоуроки Таймеры: 1 и 2

Материалы лекций:

https://github.com/ShViktor72/Education

Обратная связь:

colledge20education23@gmail.com

Задание на дом:

Задание 1: Создание простого таймера.

Разработайте приложение Windows Forms, которое использует класс **Timer** для реализации следующего функционала:

- На форме должно быть текстовое поле (Label), которое отображает текущее время (часы : минуты : секунды).
- Таймер должен обновлять время каждую секунду.
- Добавьте кнопку "Старт", которая запускает таймер, и кнопку "Стоп", которая останавливает его.

Требования:

- Используйте свойство **Interval** для установки интервала обновления времени.
- Обработайте событие **Tick** для обновления времени на форме.

Задание 2: Анимация с использованием таймера.

Разработайте приложение Windows Forms, которое создает простую анимацию с использованием таймера:

- На форме разместите кнопку (Button).
- При нажатии на кнопку "Старт" кнопка должна начать перемещаться по форме слева направо.
- Добавьте кнопку "Стоп", которая останавливает анимацию.
- Скорость перемещения кнопки должна зависеть от значения свойства **Interval**.

Требования:

- Используйте класс Timer для управления анимацией.
- Реализуйте методы **Start()** и **Stop()** для управления таймером.

Задание 3: Асинхронная задержка с обновлением прогресса.

Создайте приложение Windows Forms с кнопкой и ProgressBar.

Реализуйте асинхронный метод, который будет выполнять задержку в 5 секунд, обновляя прогресс каждые 100 миллисекунд.

При нажатии на кнопку запустите асинхронную задержку и обновляйте ProgressBar в реальном времени.

После завершения задержки выведите сообщение "Задержка завершена!".

Требования:

- Используйте Task.Delay для асинхронной задержки.
- Используйте IProgress<int> для обновления прогресса.

Задание 4: Многопоточная обработка данных.

Создайте приложение Windows Forms с кнопкой и меткой (Label).

Реализуйте асинхронный метод, который будет выполнять длительную операцию (например, суммирование чисел от 1 до 1 000 000).

При нажатии на кнопку запустите асинхронную операцию и отобразите результат в метке после завершения.

Добавьте возможность отмены операции с помощью CancellationToken.

Требования:

- Используйте Task.Run для выполнения длительной операции в фоновом потоке.
- Добавьте кнопку "Отмена", которая будет прерывать выполнение операции.

Задание 5. Таймер и движение объекта.

Создайте Windows Forms-приложение, в котором кнопка будет двигаться по форме слева направо при каждом срабатывании таймера.

Добавьте Timer на форму.

Установите интервал Timer в 100 мс.

Реализуйте обработчик события Tick, который будет изменять координаты кнопки.

При достижении правого края кнопка должна начинать движение в обратном направлении.

Дополнительное задание: реализовать изменение цвета кнопки при каждом движении.