Шпаргалка по теме "Таймеры и асинхронность в Windows Forms"

1. Класс Timer в Windows Forms

Класс Timer (System.Windows.Forms.Timer) используется для выполнения действий через определенные интервалы времени.

Основные свойства и методы Timer:

- **Interval**: Интервал времени в миллисекундах, через который будет вызываться событие Tick.
- **Enabled**: Включает (true) или выключает (false) таймер.
- **Start():** Запускает таймер.
- **Stop():** Останавливает таймер.
- **Tick**: Событие, возникающее при истечении интервала

Пример: Таймер для обновления времени на форме

```
using System;
using System.Windows.Forms;

public partial class Form1 : Form
{
    private Timer timer; // Создаем объект таймера

    public Form1()
    {
        InitializeComponent(); // Инициализация компонентов формы

        timer = new Timer(); // Создаем экземпляр таймера
        timer.Interval = 1000; // Устанавливаем интервал в 1000 мс (1 секунда)
        timer.Tick += Timer_Tick; // Подписываемся на событие Tick
        timer.Start(); // Запускаем таймер
    }

    // Обработчик события Tick (срабатывает каждую секунду)
    private void Timer_Tick(object sender, EventArgs e)
    {
        label1.Text = DateTime.Now.ToString("HH:mm:ss"); // Обновляем текст метки текущим временем
    }
}
```

Что делает код?

- Создает Timer, который срабатывает каждую секунду.
- В обработчике Tick обновляет label1, выводя текущее время в формате HH:mm:ss.
- label1 будет отображать часы, минуты и секунды, обновляясь каждую секунду.

Пример: таймер, перемещающий кнопку:

```
public partial class Form1 : Form
{
    private Timer timer; // Создаем объект таймера
    private int direction = 5; // Определяем направление движения кнопки

    public Form1()
    {
        InitializeComponent();

        // Инициализация таймера
        timer = new Timer();
        timer.Interval = 100; // Устанавливаем интервал 100 мс
        timer.Tick += Timer_Tick; // Подписываемся на событие Tick
        timer.Start(); // Запускаем таймер
    }

    private void Timer_Tick(object sender, EventArgs e)
    {
        button1.Left += direction; // Изменяем положение кнопки

        // Если кнопка достигла границы формы, меняем направление движения
        if (button1.Right >= ClientSize.Width || button1.Left <= 0)
        {
            direction = -direction;
        }
    }
}
```

2. Асинхронное программирование

Асинхронность позволяет выполнять длительные операции (загрузку данных, чтение файлов) без блокировки UI.

Ключевые слова:

- async - Обозначает, что метод может выполняться асинхронно
- await - Приостанавливает выполнение метода до завершения задачи
- Task - Представляет асинхронную задачу
- Task<T> - Асинхронная задача с возвращаемым результатом

```
// Асинхронный метод
public async Task DoWorkAsync()
{
    await Task.Delay(1000); // Асинхронная задержка
    // Продолжение работы после задержки
}

// Асинхронный метод с возвращаемым значением
public async Task<string> GetDataAsync()
{
    await Task.Delay(1000);
    return "Data";
}

// Вызов асинхронных методов
private async void button1_Click(object sender, EventArgs e)
{
    await DoWorkAsync();
    string result = await GetDataAsync();
    textBox1.Text = result;
}
```

```
// Task без возвращаемого значения
Task task = Task.Run(() =>
{
    // Длительная операция
});

// Task с возвращаемым значением
Task<int> task = Task.Run(() =>
{
    // Вычисления
    return 42;
});

// Ожидание завершения задачи
await task;

// Получение результата
int result = await task; // для Task<int>
```

Пример: Асинхронная задержка:

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

3. Асинхронные операции в Windows Forms

Пример: асинхронная загрузка данных с веб-страницы

```
private async void button1_Click(object sender, EventArgs e)
{
    using (HttpClient client = new HttpClient()) // Создаем HTTP-клиент
    {
        textBox1.Text = "Загрузка..."; // Выводим сообщение пользователю
        string data = await client.GetStringAsync("https://example.com"); // Асинхронно загружаем данные
        textBox1.Text = data; // Отображаем загруженные данные в TextBox
    }
}
```

Как это работает?

- await останавливает выполнение метода, пока GetStringAsync() загружает данные.

- UI остается отзывчивым.

Пример: Асинхронное чтение текстового файла:

```
// Асинхронный метод для чтения содержимого файла
public async Task<string> ReadFileAsync(string filePath)
{
    // Используем конструкцию using, чтобы автоматически закрыть файл после чтения
    using (StreamReader reader = new StreamReader(filePath))
    {
        // Асинхронно читаем весь файл
        string content = await reader.ReadToEndAsync();
        return content; // Возвращаем содержимое файла
    }
}

// Обработчик нажатия кнопки (асинхронный)
private async void button1_Click(object sender, EventArgs e)
{
    string filePath = "example.txt"; // Путь к файлу
    try
    {
        // Асинхронно читаем файл и ожидаем завершения операции
        string content = await ReadFileAsync(filePath);
        textBox1.Text = content; // Выводим содержимое файла в TextBox
    }
    catch (Exception ex)
    {
        // Если произошла ошибка (например, файл не найден), выводим сообщение
        MessageBox.Show("Ошибка при чтении файла: " + ex.Message, "Ошибка", MessageBoxButtons.OK, MessageBoxIcon.Error);
    }
}
```

Что делает код?

- Метод ReadFileAsync асинхронно читает файл, используя StreamReader.ReadToEndAsync().
- В button1_Click вызывается ReadFileAsync(filePath), и содержимое файла отображается в textBox1.
- В случае ошибки (например, если файл не найден) отображается MessageBox с сообщением об ошибке.

Пример: Асинхронная задержка с обновлением ProgressBar

```
// Асинхронный метод с задержкой и обновлением прогресса
public async Task DelayWithProgressAsync(IProgress<int> progress)
{
    for (int i = 0; i <= 100; i++) // Цикл от 0 до 100 (имитация прогресса)
    {
        await Task.Delay(50); // Асинхронная задержка на 50 мс
        progress.Report(i); // Сообщаем о текущем значении прогресса
    }
}

// Обработчик нажатия кнопки (асинхронный)
private async void button1_Click(object sender, EventArgs e)
{
    // Создаем объект Progress<int>, который обновляет ProgressBar
    var progress = new Progress<int>(value => progressBar1.Value = value);

    // Вызываем метод с передачей объекта прогресса
    await DelayWithProgressAsync(progress);

    // После завершения задержки показываем сообщение
    MessageBox.Show("Задержка завершена!");
}
```

Что делает код?

Метод DelayWithProgressAsync

- Запускает цикл от 0 до 100.
- Каждые 50 мс выполняет Task.Delay(50), чтобы не блокировать UI.
- Вызывает progress.Report(i), обновляя значение прогресса.

Метод button1_Click

- Создает объект Progress<int>, который обновляет progressBar1.Value.
- Вызывает DelayWithProgressAsync(progress), передавая ему объект прогресса.
- После завершения задержки выводит MessageBox с сообщением.

4. Использование Task и async/await

Пример: Асинхронная задача с возвратом результата

```
// Асинхронный метод, выполняющий длительную операцию в фоновом потоке
public async Task<int> CalculateAsync()
{
    return await Task.Run(() => // Запускаем задачу в фоновом потоке
    {
        int result = 0;
        // Длительный вычислительный процесс: суммируем числа от 0 до 999999
        for (int i = 0; i < 1000000; i++)
        {
            result += i;
        }
        return result; // Возвращаем результат
    });
}

// Обработчик нажатия кнопки (асинхронный)
private async void button1_Click(object sender, EventArgs e)
{
    int result = await CalculateAsync(); // Асинхронно вызываем метод CalculateAsync
    label1.Text = $"Результат: {result}"; // Отображаем результат в Label
}
```

Как работает код?

Метод CalculateAsync

- Запускает вычисление в фоновом потоке с помощью Task.Run().
- Выполняет сложение чисел от 0 до 999999.
- После завершения возвращает результат.

Метод button1_Click

- Асинхронно вызывает CalculateAsync(), не блокируя UI.
- После завершения вычисления обновляет label1 с результатом.

5. Отмена асинхронной операции

Пример: Использование CancellationToken

```
using System;
using System.Threading;
using System.Threading.Tasks;
using System.Windows.Forms;

public partial class Form1 : Form
{
    private CancellationTokenSource cts; // Токен для отмены операции

    public Form1()
    {
        InitializeComponent(); // Инициализация компонентов формы
    }

    // Обработчик нажатия кнопки "Старт" (асинхронный)
    private async void StartButton_Click(object sender, EventArgs e)
    {
        cts = new CancellationTokenSource(); // Создаем новый токен отмены
        try
        {
            // Запускаем длительную операцию в фоновом потоке и передаем токен отмены
            await Task.Run(() => LongRunningOperation(cts.Token), cts.Token);
        }
        catch (OperationCanceledException) // Обрабатываем отмену операции
        {
            label1.Text = "Операция отменена"; // Сообщаем пользователю об отмене
        }
    }
```

```
    // Длительная операция с возможностью отмены
    private void LongRunningOperation(CancellationToken token)
    {
        for (int i = 0; i < 100; i++)
        {
            token.ThrowIfCancellationRequested(); // Проверяем, был ли запрос на отмену
            Thread.Sleep(100); // Имитация долгого вычисления (100 * 100 мс = 10 секунд)
        }
    }

    // Обработчик нажатия кнопки "Отмена"
    private void CancelButton_Click(object sender, EventArgs e)
    {
        cts?.Cancel(); // Отправляем сигнал отмены, если токен существует
    }
}
```