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. Примеры использования асинхронных методов в Windows Forms

1. Использование компонента Timer для создания событий через интервалы времени.

Компонент **Timer** в Windows Forms — это инструмент, который позволяет выполнять определенные действия через заданные промежутки времени.

Он особенно полезен для создания приложений, где требуется периодическое выполнение кода, например, обновление интерфейса, проверка состояния или выполнение фоновых задач.

Основные характеристики компонента Timer

Событие:

- **Tick.** Основное событие компонента Timer. Оно возникает каждый раз, когда истекает заданный интервал времени. В обработчике события Tick можно разместить код, который должен выполняться периодически.

Свойства:

- **Interval**. Определяет интервал времени (в миллисекундах) между вызовами события Tick. Например, если установить Interval = 1000, событие Tick будет срабатывать каждую секунду.
- **Enabled** управляет состоянием таймера. Если Enabled = true, таймер активен и генерирует события Tick. Если Enabled = false, таймер остановлен.

Методы:

- **Start()** запускает таймер, то есть начинает отсчет времени и генерацию события Tick.
- **Stop()** останавливает таймер, прекращая выполнение события Tick.

Использование и настройка компонента Timer в конструкторе Windows Forms

1. Добавление компонента Timer на форму

Откройте ваш проект в Visual Studio.

Перейдите в режим конструктора.

Добавьте компонент Timer :

- В панели Toolbox (Область инструментов) найдите компонент Timer. Он находится в разделе Components (Компоненты).
- Перетащите компонент Timer на форму. Компонент не имеет визуального представления, поэтому он появится в нижней части конструктора (в области "непривязанных компонентов").

2. Настройка свойств Timer

После добавления компонента Timer на форму, вы можете настроить его свойства через окно Properties (Свойства). Вот основные свойства, которые нужно настроить:

- **Interval**. Установите интервал времени в миллисекундах. Например, если вы хотите, чтобы событие Tick происходило каждую секунду, установите значение 1000.
- **Enabled**. Если вы хотите, чтобы таймер сразу начал работать после запуска приложения, установите свойство Enabled в значение true. Если таймер должен быть остановлен изначально, оставьте значение false.

2. Настройка свойств Timer

После добавления компонента Timer на форму, вы можете настроить его свойства через окно Properties (Свойства). Вот основные свойства, которые нужно настроить:

- **Interval**. Установите интервал времени в миллисекундах. Например, если вы хотите, чтобы событие Tick происходило каждую секунду, установите значение 1000.
- **Enabled**. Если вы хотите, чтобы таймер сразу начал работать после запуска приложения, установите свойство Enabled в значение true. Если таймер должен быть остановлен изначально, оставьте значение false.

3. Подключение обработчика события Tick

Щелкните на компонент Timer в нижней части конструктора.

Перейдите к событиям :

- В окне Properties нажмите кнопку молнии (значок событий), чтобы перейти к списку событий.
- Найдите событие **Tick** и дважды щелкните на него. Visual Studio автоматически создаст метод-обработчик события и перенесет вас в код.

Напишите код для обработчика :

В созданном методе напишите код, который должен выполняться при каждом срабатывании таймера. Например:

```
private void timer1_Tick(object sender, EventArgs e)
{
    // Пример: Обновление текста метки
    label1.Text = DateTime.Now.ToString("HH:mm:ss");
}
```

Примечание.

Компонент System.Windows.Forms.Timer не блокирует основной интерфейс программы , если код в обработчике события Tick выполняется быстро.

Однако, если код в Tick занимает много времени (например, если вы выполняете сложные вычисления или обращаетесь к базе данных), это может привести к "заморозке" интерфейса.

Чтобы избежать этого, используйте асинхронные методы или другие типы таймеров для фоновых задач.

Для простых задач, связанных с обновлением интерфейса, System.Windows.Forms.Timer — это удобный и безопасный инструмент.

Использование компонента Timer для анимации.

Компонент Timer в Windows Forms можно эффективно использовать для создания анимации.

Анимация достигается за счет периодического изменения свойств элементов интерфейса (например, положения, размера или цвета) через заданные промежутки времени.

Как это сделать:

Компонент Timer генерирует событие Tick через заданные интервалы времени.

В обработчике события Tick изменяются свойства элемента (например, координаты или размер).

Изменения отображаются на экране, создавая эффект движения или трансформации.

Пример: Движение объекта по горизонтали.

Создадим приложение, где круг (Panel) движется в двух направлениях по форме.

Шаги:

Добавить на форму:

- Элемент Panel.
- Элемент Timer.

Написать код:

```
namespace timer11
{
    public partial class Form1 : Form
    {
        private int positionX = 0, positionY = 0; // Начальные координаты панели
        private int dx = 5, dy = 5; // Скорость движения по осям
        public Form1()
        {
            InitializeComponent();
            timer1.Interval = 25;
            timer1.Start();
        }
```

```
// Нарисуем круг в панели
private void panel1_Paint(object sender, PaintEventArgs e)
{
    Graphics g = e.Graphics;
    Brush brush = new SolidBrush(Color.Green);
    g.FillEllipse(brush, 0, 0, 100, 100);
}
```

```
// Обработчик события Tick
private void Timer1_Tick(object sender, EventArgs e)
{
    // Перемещаем панель по горизонтали и вертикали
    positionX += dx; // Изменение координаты X
    positionY += dy; // Изменение координаты Y

    // Обновляем положение панели
    panel1.Location = new Point(positionX, positionY);

    // Проверка выхода за границы формы по горизонтали
    if (positionX < 0 || positionX > this.ClientSize.Width - panel1.Width)
    {
        dx *= -1; // Меняем направление движения по горизонтали
    }

    // Проверка выхода за границы формы по вертикали
    if (positionY < 0 || positionY > this.ClientSize.Height - panel1.Height)
    {
        dy *= -1; // Меняем направление движения по вертикали
    }
}
```

В C# помимо System.Windows.Forms.Timer есть несколько других классов для работы с таймерами и выполнения кода через интервалы времени:

1. **System.Timers.Timer**. Используется для выполнения событий в фоновом потоке. Удобен для фоновых задач.

Особенности:

- Работает в отдельном потоке (может потребоваться **Invoke** при работе с UI).
- Можно использовать в консольных и сервисных приложениях.

2. **System.Threading.Timer**. Позволяет выполнять код асинхронно в другом потоке.

Особенности:

- Выполняется в потоке ThreadPool.
- Полезен для задач, не связанных с UI.

3. **System.Reactive.Linq.Observable.Interval** (Rx.NET). Позволяет создавать реактивные таймеры.

Особенности:

- Поддерживает функциональное программирование и LINQ.
- Работает асинхронно.

Если приложение Windows Forms — лучше System.Windows.Forms.Timer. Если консольное или серверное — System.Timers.Timer или System.Threading.Timer.

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:

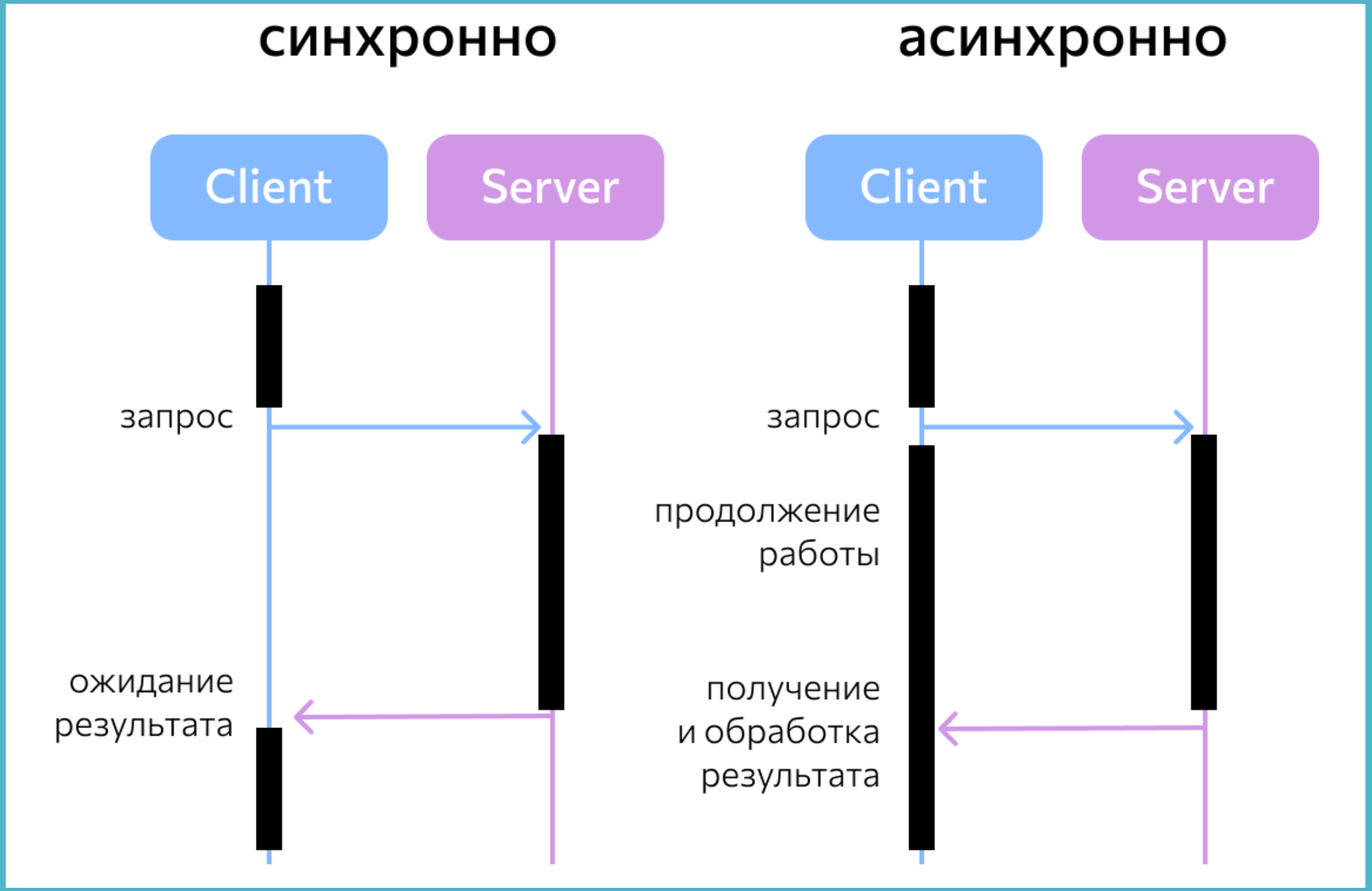
- Отрисовку пользовательского интерфейса.
- Обработку событий (нажатий кнопок, перемещения мыши и т.д.).

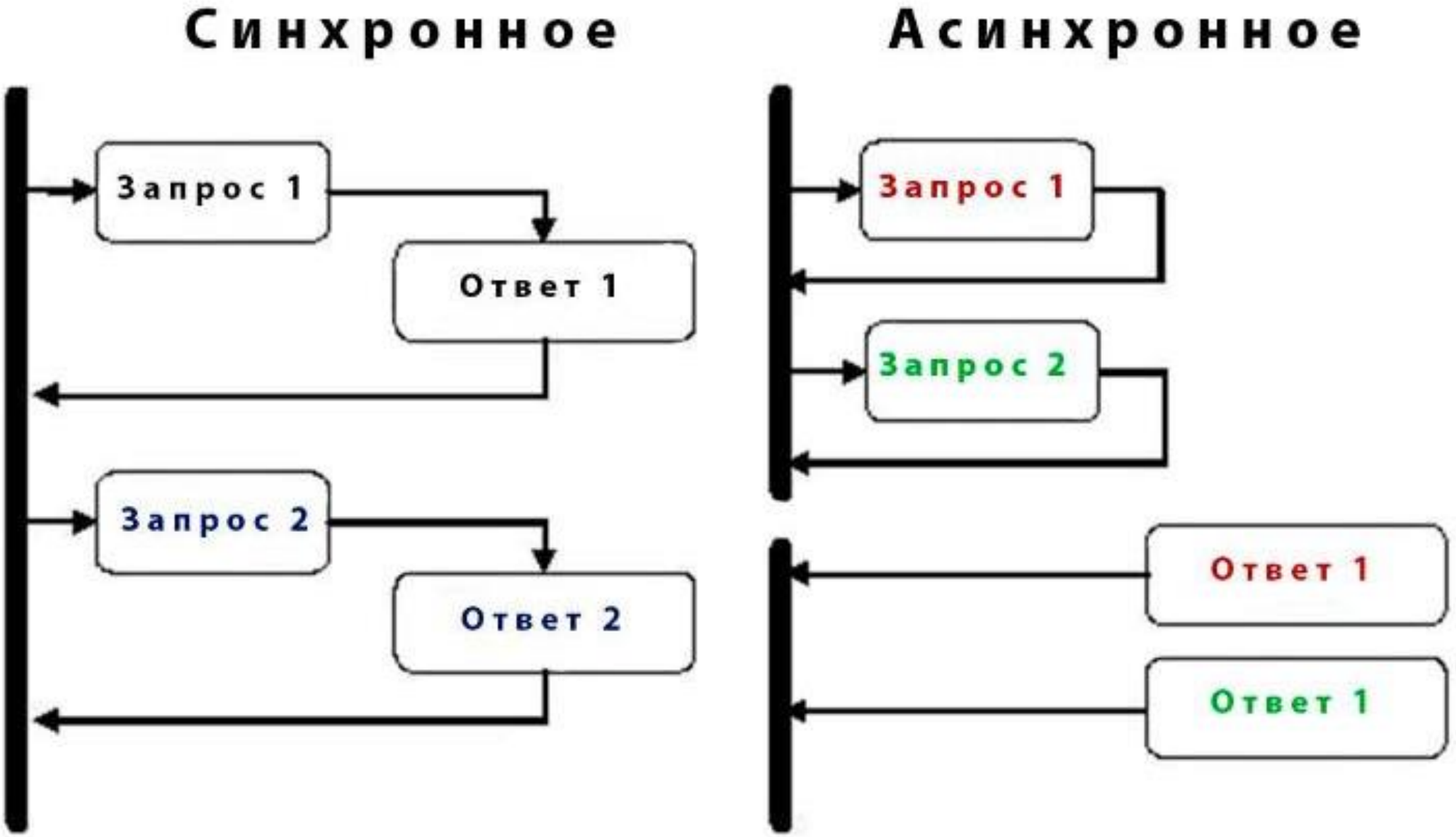
Если в этом потоке выполняется длительная операция (например, чтение файла, запрос к базе данных или загрузка данных из интернета), то:

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

Асинхронность решает эти проблемы:

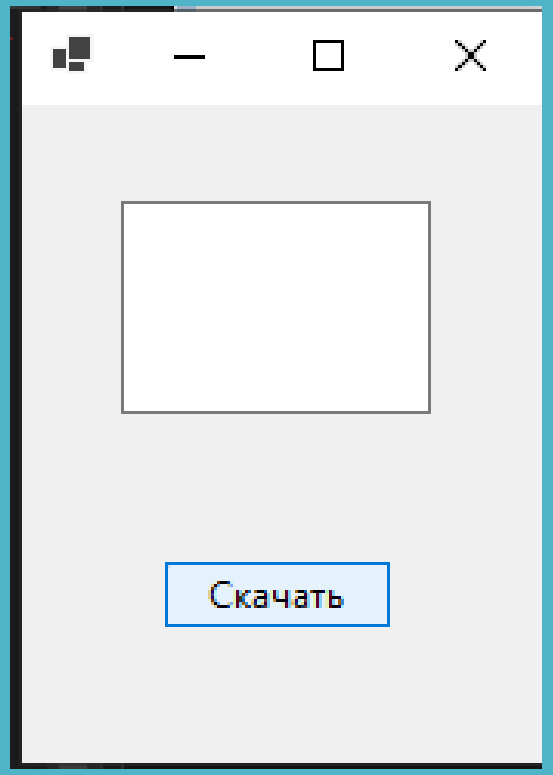
- Длительные задачи выполняются в фоне, не мешая UI.
- UI остается отзывчивым.
- Повышается производительность и удобство использования приложения.


синхронно
Client
Server
запрос
ожидание
результата
асинхронно
Client
Server
запрос
продолжение
работы
получение
и обработка
результата


Синхронное
Запрос 1
Ответ 1
Запрос 2
Ответ 2
Асинхронное
Запрос 1
Запрос 2
Ответ 1
Ответ 1

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!");
}
```

Еще примеры синхронного и асинхронного кода.

Синхронная загрузка файла:

```
// Синхронный метод загрузки
private void DownloadFile(string url, string filePath)
{
    using (HttpClient client = new HttpClient())
    {
        // Синхронный запрос для получения байтов
        byte[] fileBytes = client.GetByteArrayAsync(url).Result;

        // Сохраняем файл на диск
        File.WriteAllBytes(filePath, fileBytes);
    }
}
```

```
private void btnDowload_Click(object sender, EventArgs e)
{
    toolStripStatusLabel1.Text = "Загрузка файла...";

    try
    {
        // Выполняем синхронную загрузку файла
        DownloadFile(url, "thonny.exe");
        toolStripStatusLabel1.Text = "Файл успешно загружен!";
    }
    catch (Exception ex)
    {
        toolStripStatusLabel1.Text = $"Ошибка: {ex.Message}";
    }
}
```

Асинхронная загрузка файла:

```
// Асинхронный метод загрузки
private async Task DownloadFileAsync(string url, string filePath)
{
    using (HttpClient client = new HttpClient())
    {
        // Асинхронный запрос для получения байтов
        byte[] fileBytes = await client.GetByteArrayAsync(url);

        // Сохраняем файл на диск
        await Task.Run(() => File.WriteAllBytes(filePath, fileBytes));
    }
}
```

```
private async void btnDownAsync_Click(object sender, EventArgs e)
{
    toolStripStatusLabel1.Text = "Загрузка файла...";
    using (HttpClient client = new HttpClient())
    {
        // Асинхронный запрос для получения байтов
        byte[] fileBytes = await client.GetByteArrayAsync(url);

        // Сохраняем файл на диск
        await Task.Run(() => File.WriteAllBytes("thonny.exe", fileBytes));
        toolStripStatusLabel1.Text = "Файл успешно загружен!";
    }
}
```

3. Асинхронные операции в Windows Forms.

Существует несколько подходов к реализации асинхронных операций:

- **Использование событий и таймеров.** Подходит для простых периодических задач, которые требуют выполнения через определенные интервалы времени.
- **Использование потоков (Threads).** Дает полный контроль над фоновыми операциями. Подходит для длительных операций, таких как чтение файлов, обработка данных и запросы к базе данных.
- **Использование BackgroundWorker.** Упрощенный инструмент для выполнения задач в фоновом режиме.
- **Использование потокобезопасных библиотек**. Для сложных многопоточных приложений

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.
- 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.

4. Примеры использования асинхронных методов в Windows Forms

Пример 1: **Асинхронная загрузка файлов**

```
private async void button1_Click(object sender, EventArgs e)
{
    // Определяем URL для загрузки данных
    string url = @"https://yandex.kz";

    // Асинхронно загружаем данные с указанного URL
    string data = await DownloadAsync(url);

    // Отображаем загруженные данные в текстовом поле
    textBox1.Text = data;
}

public async Task<string> DownloadAsync(string url)
{
    // Создаем экземпляр HttpClient для выполнения HTTP-запросов
    using (HttpClient client = new HttpClient())
    {
        // Асинхронно получаем строку данных из указанного URL
        string data = await client.GetStringAsync(url);

        // Возвращаем загруженные данные
        return data;
    }
}
```

Пример 2: **Асинхронная обработка файлов**

```
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**

```
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!");
}
```

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:

1. Секундомер

Создайте приложение с кнопкой "Старт" и меткой.

При нажатии на кнопку запускается таймер, который обновляет метку каждую секунду, показывая количество прошедших секунд.

Добавьте кнопку "Стоп" для остановки таймера.

Добавьте StatusStrip для отображения состояния секундомера.

Требования:

- Используйте компонент Timer.
- Метка должна отображать время в формате "Прошло секунд: X".

2. Обратный отсчет.

Создайте приложение, которое выполняет обратный отсчет от заданного времени в секундах до 0. Когда счетчик достигает 0, выводите сообщение "Время истекло!".

Требования:

- Используйте компонент Timer.
- Используйте NumericUoDown для установки начального значения времени.
- Добавьте кнопку "Старт" для запуска отсчета.
- Оставшееся время должно отображаться в TextBox.
- После завершения отсчета таймер должен остановиться, в StatusStrip выводится соответствующее сообщение.

3. Мигающая метка

Создайте приложение с меткой, которая мигает (меняет цвет текста или фон) каждую секунду. Например, цвет текста может меняться с черного на красный и обратно.

Требования:

- Используйте компонент Timer.
- Добавьте кнопки "Старт" и "Стоп" для управления миганием.

4. Анимация движения объекта

Создайте приложение с кнопкой и прямоугольником (например, Panel). При нажатии на кнопку прямоугольник начинает двигаться по горизонтали (слева направо) с помощью таймера.

Требования:

- Используйте компонент Timer.
- Добавьте кнопки "Старт" и "Стоп" для управления движением.
- Прямоугольник должен останавливаться, если достигает края формы.

5*. 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.