C# / .NET

# Основы С# .NET

## Арифметические операции

### ^n и sqrt(n)

Библиотека Math

double square = Math.Pow(number, 2);

double squareRoot = Math.Sqrt(number);

## Ресурсы

Сайт майкрософт про всю экосистему .NET

<https://learn.microsoft.com/en-us/dotnet/>

C# language reference (ознакомление с основами языка)

<https://learn.microsoft.com/en-us/dotnet/csharp/language-reference/>

Основная документация C#

<https://learn.microsoft.com/en-us/dotnet/csharp/tour-of-csharp/>

## Типы переменных

Нельзя (string)int\_num

Нужно int\_num.ToString()

## Узнать тип объекта

object obj = "Hello";

Type type = obj.GetType();

Console.WriteLine(type.ToString()); // Вывод: System.String

int number = 42;

Console.WriteLine(number.GetType()); // Вывод: System.Int32

List<string> myList = new List<string>();

Console.WriteLine(myList.GetType()); // Вывод: System.Collections.Generic.List`1[System.String]

## Функции

### Возврат множества аргументов

private (double verminx, double verminy, double scale, double vercenx, double verceny)

ScaleEtc(Dictionary<string, List<List<double>>> vers, string ver, double maxx, double maxy)

{

double vermaxx = vers[ver][0].Max();

double verminx = vers[ver][0].Min();

double vermaxy = vers[ver][1].Max();

double verminy = vers[ver][1].Min();

double vercenx = (vermaxx + verminx) / 2;

double verceny = (vermaxy + verminy) / 2;

double verdiffx = vermaxx - verminx;

double verdiffy = vermaxy - verminy;

double scalex = maxx / verdiffx;

double scaley = maxy / verdiffy;

double scale = Math.Min(scalex, scaley);

return (verminx, verminy, scale, vercenx, verceny);

}

var (verminx, verminy, scale, vercenx, verceny) = ScaleEtc(vers, ver, maxx, maxy);

## Классы

### Класс и экземпляр класса

#### Статические методы и поля

Статические методы и поля класса отличаются от нестатических тем, что статические методы принадлежать самому классу, а не к экземпляру класса.  
  
Например метод Show()

Метод Show() делает конкретное окно (то есть экземпляр класса) видимым. Поэтому это нестатический метод.

#### this

Ключевое слово this обращается к экземпляру класса.

Пример

Обращение к полям экземпляра класса:

public partial class MainWindow : Window

{

e3Application e3; // Теперь это член экземпляра

public MainWindow(e3Application e3)

{

InitializeComponent();

this.e3 = e3; // Теперь работает корректно

}

//...

}

Обращение к полям класса

public partial class MainWindow : Window

{

static e3Application e3; // Теперь это статический член

public MainWindow(e3Application e3)

{

InitializeComponent();

MainWindow.e3 = e3; // Доступ через имя класса

}

//...

}

### Добавление функционала в методы родительского класса

public class Child : Parent

{

public override void MyMethod()

{

// Вызов родительской реализации

base.MyMethod();

// Добавление своего функционала

Console.WriteLine("Child method additional logic");

}

## Структуры данных

### Статический массив

dynamic[] devids = new dynamic[3];

### Динамический массив

### Многомерный массив

B. Многомерный массив (double[,,]):

// Объявление и создание многомерного массива

double[,,] vers2 = new double[2, 12, 12];

// Теперь vers2 - это многомерный массив double[2, 12, 12], готовый к использованию.

// Например, можно присвоить значение:

vers2[0, 0, 0] = 3.14;

### Зубчатый массив

Зубчатый массив — это структура данных в C#, которая позволяет создавать массивы массивов с разной длиной внутренних массивов. Его можно использовать для представления данных, которые могли бы быть организованы в тензор, но с одним или несколькими “измерениями” переменной длины.

A. Зубчатый массив (double[][][]):

// 1. Объявление и создание внешнего массива (первого измерения)

double[][][] vers2 = new double[2][][]; // Указываем размер только первого измерения (2)

// 2. Создание внутренних массивов (второго измерения) для каждого элемента внешнего массива

vers2[0] = new double[12][]; // Первый элемент внешнего массива содержит массив из 12 элементов

vers2[1] = new double[12][]; // Второй элемент внешнего массива содержит массив из 12 элементов

// 3. Создание самых внутренних массивов (третьего измерения) для каждого элемента второго измерения

for (int i = 0; i < 2; i++)

{

for (int j = 0; j < 12; j++)

{

vers2[i][j] = new double[12]; // Каждый элемент второго измерения содержит массив из 12 элементов

}

}

// Теперь vers2 - это зубчатый массив double[2][12][12], готовый к использованию.

// Например, можно присвоить значение:

vers2[0][0][0] = 3.14;

Мой код

double[][][][] vers2 = new double[data.dictdev.Count][][][];

for (int i = 0; i < data.dictdev.Count; i++)

{

vers2[i] = new double[10][][];

string ver = data.dictdev.Keys.ToList()[i];

foreach (string gra in data.dictdev[ver].Keys)

{

string[] gra\_ = gra.Split('\_');

int type = int.Parse(gra\_[0]);

for (int j = 0; j < data.dictdev[ver][gra].Length; j=j+2)

{

if (type == 1)

{

vers2[i][type] = new double[2][];

vers2[i][type][0] = new double[;

vers2[i][type][0][j] = data.dictdev[ver][gra][j];

vers2[i][type][1][j] = data.dictdev[ver][gra][j++];

}

}

}

}

### List<dynamic>

List<T>

<https://learn.microsoft.com/en-us/dotnet/api/system.collections.generic.list-1?view=net-9.0>

Создание динамического массива (List<T>)

using System;

using System.Collections.Generic;

class Program

{

static void Main()

{

// Создание пустого списка целых чисел

List<int> numbers = new List<int>();

// Создание списка с начальными значениями

List<string> fruits = new List<string> { "Apple", "Banana", "Cherry" };

}

}

Добавление элемента

numbers.Add(10); // Добавляет элемент 10 в конец списка

numbers.AddRange(new int[] { 20, 30, 40 }); // Добавляет несколько элементов

Удаление элемента

numbers.Remove(10); // Удаляет первое вхождение элемента 10

numbers.RemoveAt(0); // Удаляет элемент по индексу 0

numbers.RemoveRange(1, 2); // Удаляет 2 элемента, начиная с индекса 1

numbers.Clear(); // Удаляет все элементы из списка

Доступ к элементу

int firstElement = numbers[0]; // Получает элемент по индексу 0

Получение длины списка

int length = numbers.Count; // Возвращает количество элементов в списке

### Словарь

<https://learn.microsoft.com/ru-ru/dotnet/api/system.collections.generic.dictionary-2?view=net-8.0>

В C# словарь представлен структурой данных Dictionary<TKey, TValue>, где TKey — это тип ключа, а TValue — тип значения. Словарь позволяет хранить пары "ключ-значение" и обеспечивает быстрый доступ к значениям по ключу.

Создание словаря

using System;

using System.Collections.Generic;

class Program

{

static void Main()

{

// Создание пустого словаря с ключами типа string и значениями типа int

Dictionary<string, int> ages = new Dictionary<string, int>();

// Создание словаря с начальными значениями

Dictionary<string, string> capitals = new Dictionary<string, string>

{

{ "Russia", "Moscow" },

{ "USA", "Washington, D.C." },

{ "France", "Paris" }

};

}

}

Добавление элемента

ages.Add("Alice", 25); // Добавляет пару "Alice"-25 в словарь

ages["Bob"] = 30; // Альтернативный способ добавления или обновления значения

Удаление элемента

ages.Remove("Alice"); // Удаляет элемент с ключом "Alice"

ages.Clear(); // Удаляет все элементы из словаря

Доступ к элементу

int aliceAge = ages["Alice"]; // Получает значение по ключу "Alice"

Проверка наличия ключа

if (ages.ContainsKey("Alice"))

{

Console.WriteLine("Alice's age is " + ages["Alice"]);

}

Получение длины словаря

int length = ages.Count; // Возвращает количество пар "ключ-значение" в словаре

Итерация по словарю

foreach (KeyValuePair<string, int> entry in ages)

{

Console.WriteLine($"Key: {entry.Key}, Value: {entry.Value}");

}

// Или только по ключам

foreach (string key in ages.Keys)

{

Console.WriteLine($"Key: {key}");

}

// Или только по значениям

foreach (int value in ages.Values)

{

Console.WriteLine($"Value: {value}");

}

### Классы

## Многопоточность

### Инфо

<https://learn.microsoft.com/ru-ru/dotnet/api/system.threading.thread?view=net-8.0>

Console.WriteLine("Thread: " + Thread.CurrentThread.ManagedThreadId + " have started");

### Виды потоков

[STAThread] - это атрибут, который вы можете применить к методу Main вашего приложения, чтобы указать, что поток, в котором будет выполняться этот метод, должен быть создан с моделью COM-потоков Single-Threaded Apartment (STA).

Модель COM-потоков: COM (Component Object Model) - это технология Microsoft для межпроцессного взаимодействия между компонентами.

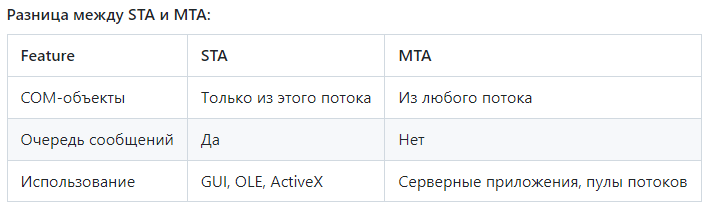
Single-Threaded Apartment (STA):

1. STA означает, что COM-объекты, созданные в этом потоке, могут вызываться только из этого же потока.
2. STA-поток имеет очередь сообщений, в которой он обрабатывает входящие вызовы от COM-объектов.
3. В контексте .NET STA-поток обычно является потоком GUI.

Устаревшие Windows API, такие как Ole, работают только в STA-потоках.

Multi-Threaded Apartment (MTA):

1. MTA означает, что COM-объекты, созданные в этом потоке, могут вызываться из любого потока.
2. MTA-потоки не имеют очереди сообщений.



#### Несколько STA потоков в проекте

В одном проекте может быть несколько STA-потоков. Это абсолютно нормально и иногда даже необходимо, особенно при работе с COM-объектами, которые требуют STA (Single-Threaded Apartment). Однако важно правильно организовать взаимодействие между этими потоками, чтобы избежать проблем с синхронизацией и утечками ресурсов.

Когда может понадобиться несколько STA-потоков?

1. Работа с несколькими независимыми COM-объектами: Если у вас есть несколько COM-объектов, которые должны работать независимо друг от друга, каждый из них может быть привязан к своему STA-потоку.
2. Изоляция задач: Если у вас есть задачи, которые могут блокировать друг друга (например, длительные операции), их можно вынести в отдельные STA-потоки.
3. Масштабируемость: Если ваше приложение должно обрабатывать несколько независимых запросов одновременно, каждый запрос может выполняться в своем STA-потоке.

Пример

using System;

using System.Threading;

using System.Windows;

namespace MultipleSTAThreads

{

public partial class App : Application

{

private Thread \_e3Thread1;

private Thread \_e3Thread2;

protected override void OnStartup(StartupEventArgs e)

{

// Инициализация первого STA-потока

\_e3Thread1 = new Thread(E3WorkerThread1)

{

IsBackground = true,

Name = "E3 STA Thread 1"

};

\_e3Thread1.SetApartmentState(ApartmentState.STA);

\_e3Thread1.Start();

// Инициализация второго STA-потока

\_e3Thread2 = new Thread(E3WorkerThread2)

{

IsBackground = true,

Name = "E3 STA Thread 2"

};

\_e3Thread2.SetApartmentState(ApartmentState.STA);

\_e3Thread2.Start();

// Инициализация главного окна

var mainWindow = new MainWindow();

mainWindow.Show();

base.OnStartup(e);

}

private void E3WorkerThread1()

{

// Инициализация COM-объекта для первого потока

dynamic e3 = Activator.CreateInstance(Type.GetTypeFromProgID("CT.Application"));

dynamic job = e3.CreateJobObject();

e3.PutMessage("E3 Thread 1 initialized");

while (true)

{

// Пример работы с COM-объектом

e3.PutMessage("Thread 1 is working");

Thread.Sleep(1000); // Имитация работы

}

}

private void E3WorkerThread2()

{

// Инициализация COM-объекта для второго потока

dynamic e3 = Activator.CreateInstance(Type.GetTypeFromProgID("CT.Application"));

dynamic job = e3.CreateJobObject();

e3.PutMessage("E3 Thread 2 initialized");

while (true)

{

// Пример работы с COM-объектом

e3.PutMessage("Thread 2 is working");

Thread.Sleep(1500); // Имитация работы

}

}

}

}

### Дeлегаты

### \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

### Основы

#### Основы управления потоками

Класс Thread

<https://learn.microsoft.com/ru-ru/dotnet/api/system.threading.thread?view=net-9.0>

System.Threading.Thread:

Это наиболее “низкоуровневый” способ создания и управления потоками.

Позволяет вам создавать новые потоки и задавать их начальный метод (ThreadStart или ParameterizedThreadStart).

Вы несете полную ответственность за управление этими потоками (запуск, завершение, синхронизация).

Пример

Thread thread1 = new Thread(thread\_e3);

Thread thread2 = new Thread(thread\_app);

thread1.Start();

thread2.Start();

thread1.Join();

thread2.Join();

#### Новый поток

new Thread(() =>

{

Console.WriteLine("Thread: " + Thread.CurrentThread.ManagedThreadId + " have started");

}).Start();

#### Функция для нового потока

using System.Threading;

namespace DuoProcc

{

public class Program

{

public static void Main()

{

Thread workerThread = new Thread(() => WorkerThreadProc(e3, job));

workerThread.Start();

workerThread.Join();

}

}

}

static void WorkerThreadProc(dynamic e3, dynamic job)

{

//something

}

#### Экстренная остановка потока

#### Правильная остановка потока

<https://www.google.com/search?q=cancellationtokensource+tokensource+youtube&sca_esv=0fd6655462c7c58e&rlz=1C1BNSD_ruRU1144RU1144&ei=sxqaZ6TVFseE1fIPra6ySA&oq=CancellationTokenSource+tokenSource+you&gs_lp=Egxnd3Mtd2l6LXNlcnAiJ0NhbmNlbGxhdGlvblRva2VuU291cmNlIHRva2VuU291cmNlIHlvdSoCCAAyBxAhGKABGAoyBxAhGKABGAoyBxAhGKABGApI7ypQgAFY-CJwBHgBkAEAmAGlAaABvQWqAQM1LjK4AQPIAQD4AQGYAgugAuIFwgIKEAAYsAMY1gQYR8ICCRAhGKABGAoYKpgDAIgGAZAGCJIHAzguM6AHoSA&sclient=gws-wiz-serp#fpstate=ive&vld=cid:3af15844,vid:TKc5A3exKBQ,st:0>

using System;

using System.Threading;

CancellationTokenSource tokenSource = new(); // Create a token source.

Thread thread = new(

() => LongRunningOperation(tokenSource.Token) // Pass the token to the thread you want to stop.

);

thread.Start();

Thread.Sleep(1500);

tokenSource.Cancel(); // Request cancellation.

thread.Join(); // If you want to wait for cancellation, `Join` blocks the calling thread until the thread represented by this instance terminates.

tokenSource.Dispose(); // Dispose the token source.

Thread.Sleep(1500);

void LongRunningOperation(CancellationToken token)

{

while(!token.IsCancellationRequested) { // Check if the caller requested cancellation.

Console.WriteLine("I'm running");

Thread.Sleep(500);

}

}

Или

void LongRunningOperation(CancellationToken token)

{

while(\*Condition\*)

{

Console.WriteLine("I'm running");

Thread.Sleep(500);

}

If token.Canseled

{

\* do something

}

### (Очередь) BlockingCollection<T>

Инструмент:

Очередь задач в потоке

BlockingCollection<T> \_e3Queue = new BlockingCollection<T>();

<https://learn.microsoft.com/ru-ru/dotnet/api/system.collections.concurrent.blockingcollection-1?view=net-8.0>

Аргументы:

Для передачи обычной функции в качестве задачи используется

BlockingCollection<Action> \_e3Queue = new BlockingCollection<Action>();

Если нужно выполнить какое-то действие (например, вывести сообщение в консоль) в отдельном потоке, использование Action является абсолютно правильным и подходящим выбором.

1. Представляет собой действие: Action - это делегат, который представляет метод, не принимающий аргументов и не возвращающий значений, что точно соответствует вашей потребности выполнить некое действие (вывод в консоль).
2. Простота и лаконичность: Использование Action позволяет вам легко передавать и выполнять анонимные методы (лямбды), что делает ваш код более читаемым и компактным.

Принцип очереди BlockingCollection<Т>:

1. Если очередь пуста: Метод GetConsumingEnumerable() блокирует поток до тех пор, пока в очереди не появится новая задача.Поток переходит в состояние ожидания и не потребляет ресурсы процессора.
2. Если очередь завершена: Если вызван метод CompleteAdding() для BlockingCollection<T>, то GetConsumingEnumerable() завершает работу, и поток выходит из цикла foreach. После этого поток завершает выполнение метода E3WorkerThread() и завершается.

#### Другие типы T

Другие типы делегатов и их отличия:

1. Action – функция без параметров и возвращаемых значений
2. Action<T> (с параметрами): Этот делегат представляет метод, который принимает один или несколько параметров, но не возвращает значения.
   * Action<int>: Метод принимает один параметр типа int.
   * Action<string, int>: Метод принимает два параметра типа string и int.
3. Action<string> greet = (name) => Console.WriteLine($"Hello, {name}!");

greet("Alice"); *// Передача параметра*

Csharp

1. Func<TResult> (с возвращаемым значением): Этот делегат представляет метод, который не принимает никаких параметров, но возвращает значение типа TResult.
2. Func<int> getRandomNumber = () => new Random().Next(100);

int number = getRandomNumber(); *// Получение возвращаемого значения*

Csharp

1. Func<T, TResult> (с параметром и возвращаемым значением): Этот делегат представляет метод, который принимает один или несколько параметров, и возвращает значение.
   * Func<int, bool>: Метод принимает int и возвращает bool.
   * Func<int, int, string>: Метод принимает два int и возвращает string.
2. Func<int, int> square = (x) => x \* x;

int squaredValue = square(5); *// Получение результата*

Csharp

1. Predicate<T>: Представляет метод, который принимает один параметр типа T и возвращает bool. Обычно используется для фильтрации элементов в коллекциях.
2. Predicate<int> isEven = (number) => number % 2 == 0;

bool result = isEven(4); *// Вызов предиката, result будет true*

#### Контекстные переменные

##### !!!!!!!!!!!Захват контекста с помощью аргументов функции

#### \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

#### Простой пример

public static void Main(string[] args)

{

Console.WriteLine("Hello");

BlockingCollection<Action> queue = new BlockingCollection<Action>();

Thread worker = new Thread(() => Worker(queue));

worker.Start();

int i = 1;

Thread.Sleep(2000);

queue.Add(() => { Console.WriteLine(Thread.CurrentThread.ManagedThreadId + " is working on "+ i++); });

queue.Add(() => { Console.WriteLine(Thread.CurrentThread.ManagedThreadId + " is working on " + i++); });

Thread.Sleep(2000);

queue.Add(() => { Console.WriteLine(Thread.CurrentThread.ManagedThreadId + " is working on " + i++); });

Thread.Sleep(1000);

queue.Add(() => { Console.WriteLine(Thread.CurrentThread.ManagedThreadId + " is working on " + i++); });

// Завершение работы потока-воркера

queue.CompleteAdding();

worker.Join();

Console.WriteLine("Finished");

}

private static void Worker(BlockingCollection<Action> queue)

{

Console.WriteLine("Thread ID: " + Thread.CurrentThread.ManagedThreadId);

foreach (var action in queue.GetConsumingEnumerable())

{

action(); // Выполняем задачу в потоке E3

}

}

}

#### Поток для выполнения очереди задач

// App.xaml.cs

public partial class App : Application

{

// Очередь задач для E3 потока

private BlockingCollection<Action> \_e3Queue = new BlockingCollection<Action>();

private dynamic \_e3;

private dynamic \_job;

// Поток E3

private Thread \_e3Thread;

protected override void OnStartup(StartupEventArgs e)

{

// Инициализация потока E3

\_e3Thread = new Thread(E3WorkerThread)

{

IsBackground = true,

Name = "E3 STA Thread"

};

\_e3Thread.SetApartmentState(ApartmentState.STA); // Критически важно для COM

\_e3Thread.Start();

// Инициализация главного окна

var mainWindow = new MainWindow();

mainWindow.Show();

base.OnStartup(e);

}

private void E3WorkerThread()

{

// Инициализация COM-объектов в своем потоке

\_e3 = Activator.CreateInstance(Type.GetTypeFromProgID("CT.Application"));

\_job = \_e3.CreateJobObject();

\_e3.PutMessage("E3 initialized");

// Обработка задач из очереди

foreach (var action in \_e3Queue.GetConsumingEnumerable())

{

action(); // Выполняем задачу в потоке E3

}

}

// Метод для постановки задач в очередь E3

public void PostToE3(Action action)

{

\_e3Queue.Add(action);

}

protected override void OnExit(ExitEventArgs e)

{

// Корректное завершение потока

\_e3Queue.CompleteAdding();

\_e3Thread.Join(5000);

base.OnExit(e);

}

}

// MainWindow.xaml.cs

public partial class MainWindow : Window

{

public MainWindow()

{

InitializeComponent();

// Подключаем обработчик кликов (для примера)

this.MouseDown += MainWindow\_MouseDown;

}

private void MainWindow\_MouseDown(object sender, MouseButtonEventArgs e)

{

if (e.ChangedButton == MouseButton.Left)

{

// Ставим задачу в очередь E3

((App)Application.Current).PostToE3(ProcessLeftClick);

}

}

private void ProcessLeftClick()

{

try

{

var app = (App)Application.Current;

// Работаем с COM-объектами в потоке E3

dynamic[] devids = new dynamic[3];

int cnt = app.\_job.GetSelectedDeviceIds(ref devids);

if (cnt > 0)

{

app.\_e3.PutMessage($"Selected device: {devids[1]}");

// Обновление UI через Dispatcher

Dispatcher.Invoke(() =>

{

Title = $"Last ID: {devids[1]}"; // Обновляем заголовок окна

});

}

}

catch (Exception ex)

{

Dispatcher.Invoke(() => MessageBox.Show(ex.Message));

}

}

}

# Подключение проектов

Перед тем как подключать проекты НУЖНО убедится, что они

1. Предназначены для одной платформы (windows, any, etc)
2. Имеют один фреймворк(.NET core / .NET / .NET Framework) ЭТО ВСЁ РАЗНЫЕ ФРЕЙМВОРКИ
3. Имеют одну версию фреймворка.

## Подключение csproj

Имея готовый проект, его можно подключить к другому проекту, но они должны находится в ОДНОМ SOLUTION

Нужно

1. В решении (solution) добавить или создать нужный проект
2. Добавить в исходном проекте в зависимости ссылку на добавленный проект
3. И в исходном проект написать using <namespace of added proj>
4. Теперь можно использовать классы и типы добавленного проекта.

## Подключить DLL

Имея исходный проект можно подключить классы и типы данных нужного проекта извне, то нужно забилдить его как DLL и подключить

## Через exe

using System.Diagnostics;

namespace test1

{

class Program

{

[STAThread]

public static void Main()

{

string wpfAppPath = @"D:\#work\zadaniya\1 изображения\test2\WPF Interface\WpfApp1\bin\Debug\WpfApp1.exe";

Process.Start(wpfAppPath);

}

}

}

# Advanced C#

## Создание наследников абстрактного класса

### Вручную

public class Graphelement

{

public string Name { get; set; }

public string Charr { get; set; }

}

public class Circle : Graphelement

{

public double Radius { get; set; }

}

public class Rectangle : Graphelement

{

public double Width { get; set; }

public double Height { get; set; }

}

public static class GraphelementFactory

{

public static Graphelement CreateGraphelement(string type, string name, string charr, params object[] additionalParams)

{

switch (type.ToLower())

{

case "circle":

if (additionalParams.Length < 1) throw new ArgumentException("Недостаточно параметров для создания круга");

return new Circle { Name = name, Charr = charr, Radius = (double)additionalParams[0] };

case "rectangle":

if (additionalParams.Length < 2) throw new ArgumentException("Недостаточно параметров для

создания прямоугольника");

return new Rectangle { Name = name, Charr = charr,

Width = (double)additionalParams[0],

Height = (double)additionalParams[1] };

default:

throw new ArgumentException("Неизвестный тип графического элемента");

}

}

}

class Program

{

static void Main()

{

// Создание круга

var circle = GraphelementFactory.CreateGraphelement("circle", "Circle1", "C", 5.0);

Console.WriteLine($"Circle: {circle.Name}, {circle.Charr}, Radius: {((Circle)circle).Radius}");

// Создание прямоугольника

var rectangle = GraphelementFactory.CreateGraphelement("rectangle", "Rectangle1", "R", 10.0, 20.0);

Console.WriteLine($"Rectangle: {rectangle.Name}, {rectangle.Charr}, Width: {((Rectangle)rectangle).Width}, Height: {((Rectangle)rectangle).Height}");

}

}

### Автоматическое создание новых классов наследников

using System;

using System.Collections.Generic;

public class Graphelement

{

public string Name { get; set; }

public string Charr { get; set; }

}

public class Circle : Graphelement

{

public double Radius { get; set; }

}

public class Rectangle : Graphelement

{

public double Width { get; set; }

public double Height { get; set; }

}

public static class GraphelementFactory

{

private static readonly Dictionary<string, Func<string, string, object[], Graphelement>> \_creators =

new Dictionary<string, Func<string, string, object[], Graphelement>>

{

{ "circle", (name, charr, additionalParams) =>

{

if (additionalParams.Length < 1) throw new ArgumentException("Недостаточно параметров для создания круга");

return new Circle { Name = name, Charr = charr, Radius = (double)additionalParams[0] };

}

},

{ "rectangle", (name, charr, additionalParams) =>

{

if (additionalParams.Length < 2) throw new ArgumentException("Недостаточно параметров для создания прямоугольника");

return new Rectangle { Name = name, Charr = charr, Width = (double)additionalParams[0], Height = (double)additionalParams[1] };

}

}

};

public static void RegisterType(string type, Func<string, string, object[], Graphelement> creator)

{

\_creators[type.ToLower()] = creator;

}

public static Graphelement CreateGraphelement(string type, string name, string charr, params object[] additionalParams)

{

if (\_creators.TryGetValue(type.ToLower(), out var creator))

return creator(name, charr, additionalParams);

throw new ArgumentException("Неизвестный тип графического элемента");

}

}

class Program

{

static void Main()

{

// Создание круга

var circle = GraphelementFactory.CreateGraphelement("circle", "Circle1", "C", 5.0);

Console.WriteLine($"Circle: {circle.Name}, {circle.Charr}, Radius: {((Circle)circle).Radius}");

// Создание прямоугольника

var rectangle = GraphelementFactory.CreateGraphelement("rectangle", "Rectangle1", "R", 10.0, 20.0);

Console.WriteLine($"Rectangle: {rectangle.Name}, {rectangle.Charr}, Width: {((Rectangle)rectangle).Width}, Height: {((Rectangle)rectangle).Height}");

// Регистрация нового типа

GraphelementFactory.RegisterType("triangle", (name, charr, additionalParams) =>

{

if (additionalParams.Length < 3) throw new ArgumentException("Недостаточно параметров для создания треугольника");

return new Triangle { Name = name, Charr = charr, SideA = (double)additionalParams[0], SideB = (double)additionalParams[1], SideC = (double)additionalParams[2] };

});

// Создание треугольника

var triangle = GraphelementFactory.CreateGraphelement("triangle", "Triangle1", "T", 3.0, 4.0, 5.0);

Console.WriteLine($"Triangle: {triangle.Name}, {triangle.Charr}, Sides: {((Triangle)triangle).SideA}, {((Triangle)triangle).SideB}, {((Triangle)triangle).SideC}");

}

}

public class Triangle : Graphelement

{

public double SideA { get; set; }

public double SideB { get; set; }

public double SideC { get; set; }

}

# E3.Series

## Подключение к работающему приложению е3

1. В зависимостях подключить СOM класс е3 к проекту
2. Код

using e3;

dynamic type = Type.GetTypeFromProgID("CT.Application");

e3Application e3 = Activator.CreateInstance(type);

dynamic job = e3.CreateJobObject();

e3.PutMessage("Hello E3 from .NET");

## Все устройства в проекте

dynamic[] devids = new dynamic[3];

int cntdev = job.GetDeviceIds(ref devids);

dynamic dev = job.CreateDeviceObject();

job.GetSelectedDeviceIds(ref devids);

e3.PutMessage("count of devices: " + cntdev);

for (int i = 1; i< devids.Length; i++)

{

dev.SetId(devids[i]);

e3.PutMessage(devids[i]);

}

# WPF

## Документация

Документацию можно найти на сайте .NET

<https://learn.microsoft.com/ru-ru/dotnet/?view=windowsdesktop-9.0>

по классу **using System.Windows;**

[**https://learn.microsoft.com/ru-ru/dotnet/api/?preserve-view=true**](https://learn.microsoft.com/ru-ru/dotnet/api/?preserve-view=true)

“[System.Windows](https://learn.microsoft.com/ru-ru/dotnet/api/system.windows) Namespace”

<https://learn.microsoft.com/ru-ru/dotnet/api/system.windows?view=windowsdesktop-9.0>

## Инфо

System.Windows.Window в API

## Сравнение с WinForms

Почему я могу с помощью классов windows forms мы можем получать доступ к событиям за пределами окон? wpf предоставляет доступ только к оконным событиям winforms получается более гибкий инструмент (хоть у него меньше интсрументов итд)

Ваш вопрос касается ключевых различий между Windows Forms (WinForms) и Windows Presentation Foundation (WPF) в контексте обработки событий, особенно за пределами окон приложения. Давайте разберемся, почему WinForms может казаться более гибким в этом аспекте, и почему WPF ограничен в доступе к событиям за пределами окон.

1. **Архитектурные различия между WinForms и WPF**

WinForms:

**Основан на WinAPI**: WinForms — это обертка над WinAPI (Windows API), которая предоставляет доступ к низкоуровневым функциям операционной системы. Это позволяет WinForms работать с глобальными событиями, такими как клики мыши или нажатия клавиш, даже за пределами окон приложения.

**Глобальные хуки**: WinForms может использовать системные хуки (например, через System.Windows.Forms или SetWindowsHookEx из WinAPI), чтобы отслеживать события на уровне всей операционной системы.

**Простота и прямое взаимодействие**: WinForms был разработан для быстрой разработки приложений с прямым доступом к системным ресурсам, что делает его более гибким для задач, связанных с глобальными событиями.

WPF:

**Абстракция над DirectX**: WPF построен на основе DirectX и использует собственную систему рендеринга. Это делает его более мощным для создания сложных пользовательских интерфейсов, но также добавляет уровень абстракции, который ограничивает прямой доступ к низкоуровневым системным событиям.

**Ориентация на безопасность и изоляцию**: WPF был разработан с акцентом на безопасность и изоляцию приложения от системных ресурсов. Это означает, что WPF не предоставляет встроенных механизмов для работы с глобальными событиями, такими как клики мыши за пределами окон приложения.

**События только в пределах окон**: WPF обрабатывает события (например, клики мыши или нажатия клавиш) только в пределах окон приложения. Это связано с тем, что WPF фокусируется на управлении пользовательским интерфейсом, а не на взаимодействии с системой на низком уровне.

2. **Почему WinForms кажется более гибким?**

WinForms действительно может казаться более гибким, потому что:

**Прямой доступ к WinAPI**: WinForms позволяет использовать низкоуровневые функции Windows, такие как глобальные хуки, что дает больше контроля над системными событиями.

**Меньше абстракций**: WinForms ближе к операционной системе, что упрощает доступ к системным ресурсам и событиям.

**Историческая причина**: WinForms был разработан раньше WPF и изначально предназначался для создания приложений, которые тесно взаимодействуют с операционной системой.

3. **WPF и его ограничения**

WPF был разработан с другими целями:

**Абстракция и безопасность**: WPF предоставляет более высокоуровневую абстракцию, что делает его более безопасным, но менее гибким для задач, требующих низ

## Подключения проектов

К WPF приложению нельзя подключить консольный проект !!!!!!!!

## \_\_\_\_\_\_\_\_\_Основы\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

## Xaml

### Закругление краёв окна

<Window x:Class="Wpf\_interface.MainWindow"

xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"

xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"

xmlns:d="http://schemas.microsoft.com/expression/blend/2008"

xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"

xmlns:local="clr-namespace:Wpf\_interface"

WindowStyle="None"

mc:Ignorable="d"

Topmost="True"

Title="MainWindow"

Height="200"

Width="70"

AllowsTransparency="True"

Background="Transparent">

<Border CornerRadius="4 4 23 23" BorderThickness="0.5" BorderBrush="Black" Background="White">

<Grid>

<TextBox x:Name="txt" HorizontalAlignment="Center" Margin="0,32,0,0" TextWrapping="Wrap" Text="" VerticalAlignment="Top" Width="48"/>

</Grid>

</Border>

</Window>

### Text box

## App.xaml.cs

Класс Application документация

<https://learn.microsoft.com/ru-ru/dotnet/api/system.windows.application?view=windowsdesktop-9.0>

App.cs – является бесконечным циклом main, в котором может работать то или иное окно.

Пока существует (не закрыт) хотя бы один экземпляр window – run будет продолжать бесконечный цикл, завершится только после закрытия этого экземпляра.

MainWindows.cs – само окно, и при window.Show() – показывает окно на 1 секунду.

### Управление мейном приложеня

<https://learn.microsoft.com/ru-ru/dotnet/api/system.windows.application?view=windowsdesktop-9.0#properties>

Изменяя параметр shutdownMode в app.xaml можно изменить то, когда приложение завершит работу

<Application x:Class="WpfApp1.App"

xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"

xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"

xmlns:local="clr-namespace:WpfApp1"

ShutdownMode="OnExplicitShutdown"

StartupUri="MainWindow.xaml">

<Application.Resources>

</Application.Resources>

</Application>

OnLastWindowClose 0

An application shuts down when either the last window closes, or Shutdown() is called.

OnMainWindowClose 1

An application shuts down when either the main window closes, or Shutdown() is called.

OnExplicitShutdown 2

An application shuts down only when Shutdown() is called.

## Window.xaml.cs

### Base window

Window.xaml.cs

using System.Text;

using System.Windows;

using System.Windows.Controls;

using System.Windows.Data;

using System.Windows.Documents;

using System.Windows.Input;

using System.Windows.Media;

using System.Windows.Media.Imaging;

using System.Windows.Navigation;

using System.Windows.Shapes;

namespace Wpf\_interface

{

/// <summary>

/// Interaction logic for MainWindow.xaml

/// </summary>

public partial class MainWindow : Window

{

public MainWindow()

{

InitializeComponent();

}

}

}

Window.xaml

<Window x:Class="Wpf\_interface.MainWindow"

xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"

xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"

Title="MainWindow" Height="450" Width="800">

<Grid>

<TextBlock HorizontalAlignment="Center" VerticalAlignment="Center">

Hello, WPF!

</TextBlock>

</Grid>

</Window>

### Получение координат мыши

## Элементы

### Grid

#### Создание grid

public partial class MainWindow : Window

{

Grid grid;

public MainWindow()

{

InitializeComponent();

// Создаем новый Grid

grid = new Grid();

grid.ColumnDefinitions.Add(new ColumnDefinition());

this.Content = grid;

#### Настройка grid

grid = new Grid();

grid.ColumnDefinitions.Add(new ColumnDefinition{Width =new GridLength(columnHeight) });

grid.RowDefinitions.Add(new RowDefinition { Height = new GridLength(rowHeight) });

### Button

Button button = new Button

{

Content = "Click Me" + i.ToString(),

Width = 100, // Устанавливаем фиксированную ширину

Height = 30, // Устанавливаем фиксированную высоту

//HorizontalAlignment = HorizontalAlignment.Center,

//VerticalAlignment = VerticalAlignment.Center

HorizontalAlignment = HorizontalAlignment.Left, // Прижимаем к левому краю

VerticalAlignment = VerticalAlignment.Top, // Прижимаем к верхнему краю

Margin = new Thickness(10, 5, 0, 0) // Отступ слева (10) и сверху (5), 0 - отступы справа и снизу

};

Grid.SetRow(button, rowCount - 1);

Grid.SetColumn(button, columnCount - 1);

grid.Children.Add(button);

### Окно

[System.Windows](https://learn.microsoft.com/ru-ru/dotnet/api/system.windows?view=windowsdesktop-9.0).Window

<https://learn.microsoft.com/ru-ru/dotnet/api/system.windows.window?view=windowsdesktop-9.0>

Создание окна

MainWindow window = new MainWindow();

window.Show();

Удаление окна (очищение памяти)

window.Close();

#### Свойства

1. Topmost

Окно всегда будет поверх других окн в системе виндовс пользователя

Topmost

1. добавление в xaml файл окна

<Window x:Class="Wpf\_interface.MainWindow"

xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"

xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"

xmlns:d="http://schemas.microsoft.com/expression/blend/2008"

xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"

xmlns:local="clr-namespace:Wpf\_interface"

mc:Ignorable="d"

Topmost="True"

Title="MainWindow" Height="450" Width="800">

1. изменение параметра через код

using System.Windows;

namespace YourNamespace

{

public partial class YourWindow : Window

{

public YourWindow()

{

InitializeComponent();

this.Topmost = true; // Устанавливаем Topmost в true

}

}

}

1. Visibility

<https://learn.microsoft.com/ru-ru/dotnet/api/system.windows.uielement.visibility?view=windowsdesktop-9.0#system-windows-uielement-visibility>

Пример

private void PlayMedia(object sender, MouseButtonEventArgs args)

{

pauseBTN.Visibility = Visibility.Visible;

playBTN.Visibility = Visibility.Collapsed;

media.SpeedRatio = 1.0;

media.Play();

}

## Canvas ( черчение объектов)

System.Windows.Controls.Canvas

<https://learn.microsoft.com/ru-ru/dotnet/api/system.windows.controls.canvas?view=windowsdesktop-9.0>

Объекты которые можно добавить

System.Windows.Shapes

### Начертить линию

using System.Windows;

using System.Windows.Controls;

using System.Windows.Media;

using System.Windows.Shapes;

namespace WpfApp

{

public partial class MainWindow : Window

{

public MainWindow()

{

InitializeComponent();

// Создаем Canvas

var canvas = new Canvas();

this.Content = canvas;

// Создаем линию

var line = new Line

{

X1 = 50,

Y1 = 50,

X2 = 200,

Y2 = 200,

Stroke = Brushes.Black,

StrokeThickness = 2

};

// Добавляем линию на Canvas

canvas.Children.Add(line);

}

}

}

## Управление сторонним приложением (Background thread)

### !!!Управление потоками

При подключении и запуске wpf сторонним приложением исходный программный поток заходит в wpf и остается там, пока окно не закроется.

Поэтому для управления этим окном нужно создавать второй поток ДО запуска приложения.

Пример:

Так не работает!!

[STAThread]

public static void Main()

{

var app = new App();

var window = new MainWindow();

app.Run(window);

window.labelcontent = "HELLO WPF from Console!!!";

Даже с диспатчером не работает, так как окно будет удалено на момент строчки window.labelcontent = "HELLO WPF from Console!!!";

Нужно вот так:

namespace Script

{

class Program

{

[STAThread]

public static void Main()

{

Console.WriteLine("Thread: " + Thread.CurrentThread.ManagedThreadId + " have started");

Thread.Sleep(1000);

var app2 = new WpfApp2.App();

var window2 = new WpfApp2.MainWindow();

new Thread(() =>

{

Console.WriteLine("Thread: " + Thread.CurrentThread.ManagedThreadId + " have started");

Thread.Sleep(3000);

window2.txttext = "HELLO WPF from Console!!!";

Console.WriteLine("Thread: " + Thread.CurrentThread.ManagedThreadId + " have ended");

}).Start();

app2.Run(window2);

Console.WriteLine("Thread: " + Thread.CurrentThread.ManagedThreadId + " have ended");

### Запуск из стороннего приложения (консольного например)

1. Добавление или создание нового приложения WPF в существующем проекте
2. Подключение WPF в зависимостях исходного приложения
3. Код

В коде ниже wpf приложение запустится в потоке самого скрипта, поэтому если нужно что то делать паралельно с работой WPF, нужно создавать новый поток до app.Run()

using System.Reflection.Emit;

using Wpf\_interface;

namespace Script

{

class Program

{

[STAThread]

public static void Main()

{

var app = new App();

var window = new MainWindow();

app.Run(window);

}

}

}

### Управление мейном приложения из сторонней программы

Чтобы полностью контролировать main (бесконечный цикл app.Run())

В app.xaml

1. Убрать свойство startUri (чтобы убрать дефолтное окно в начале работы app.Run и создавать окна самому когда нужно)
2. Установить свойство OnExplicitExit

<Application x:Class="shutdown.App"

xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"

xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"

xmlns:local="clr-namespace:shutdown"

ShutdownMode="OnExplicitShutdown">

<Application.Resources>

</Application.Resources>

</Application>

Теперь можно самому в своем скрипте управлять мейном WPF

В script.cs

namespace Shutdown\_console

{

class Programm

{

[STAThread]

static void Main()

{

App app = new App();

app.InitializeComponent();

// process //

app.Run();

app.Shutdown();

}

}

}

### Создание окон из другого треда

<https://learn.microsoft.com/en-us/dotnet/desktop/wpf/advanced/threading-model?view=netframeworkdesktop-4.8>

#### Кратко

App.cs

public partial class App : Application

{

public MainWindow SameThreadWindow2()

{

MainWindow window = null;

Dispatcher.Invoke(() =>

{

MainWindow window = new MainWindow();

window.Show();

});

return window;

}

}

Script.cs

class Program

{

[STAThread]

static void Main()

{

App app = new App();

Thread workerThread = new Thread(() => thread\_e3(app));

workerThread.Start();

app.Run();

}

static public void thread\_e3(App app)

{

Thread.Sleep(3000);

MainWindow window = app.SameThreadWindow2();

}

}

#### Полно

App.cs

\_\_\_\_\_\_\_\_\_\_

public partial class App : Application

{

private void ThreadStartingPoint()

{

new MainWindow().Show();

System.Windows.Threading.Dispatcher.Run();

}

public void SameThreadWindow2()

{

Dispatcher.Invoke(() =>

{

new MainWindow().Show();

});

}

public void NewThreadWindow2()

{

Dispatcher.Invoke(() =>

{

Thread newWindowThread = new Thread(ThreadStartingPoint);

newWindowThread.SetApartmentState(ApartmentState.STA);

newWindowThread.IsBackground = true;

newWindowThread.Start();

});

}

}

}

Script.cs

\_\_\_\_\_\_\_\_\_\_\_

class Program

{

[STAThread]

static void Main()

{

App app = new App();

Thread workerThread = new Thread(() => thread\_e3(app));

workerThread.Start();

app.Run();

}

static public void thread\_e3(App app)

{

Thread.Sleep(3000);

app.SameThreadWindow2();

app.NewThreadWindow2();

}

}

### Передача переменных из стороннего приложения

Для изменения свойств класса окна требуется понимать:

1. Изменить свойство класса окна или свойства элементов окна нельзя ЗА ПРЕДЕЛАМИ класса, они все protected и даже если заменить на public – это будет выдавать ошибку. Правильно нужно создать ПОЛЬЗОВАТЕЛЬСКОЕ свойство, которое бы внутри класса выполняло бы замену. Вот реализация такого свойства.

public string txttext

{

get { return (string)txt.Text; }

set { txt.Text = value; }

}

1. WPF устроено таким образом, что изменение свойств и выполнения функций класса нашего окна может быть выполнено только в ГЛАВНОМ ПОТОКЕ, который создал это окно.

Поэтому для изменения свойств из ДРУГОГО потока нужно ЗАЙТИ в главный поток и выполнить нужные изменения в главном потоке.

Реализация:

Dispatcher.Invoke(() =>

{

txt.Text = value;

});

\*Также важно понимать, что если создать событие на которое будет далее происходить что то – ЭТО ТАКЖЕ НОВЫЙ ПОТОК и он не сможет изменить свойства.

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Таким образом:

Вот новый поток, который будет изменять свойства

new Thread(() =>

{

Console.WriteLine("Thread: " + Thread.CurrentThread.ManagedThreadId + " have started");

Thread.Sleep(3000);

window2.txttext = "HELLO WPF from Console!!!";

Console.WriteLine("Thread: " + Thread.CurrentThread.ManagedThreadId + " have ended");

}).Start();

// ВНУТРИ КЛАССА ОКНА //

public string txttext

{

get { return (string)txt.Text; }

set

{

Dispatcher.Invoke(() =>

{

txt.Text = value;

});

}

}

## Низкоуровневые события

За пределами окон wpf редко когда может предоставить доступ к каким то событиям (клик мыши за пределами окна, координаты мыши за пределами окна итд), особенно когда окно ещё даже не создано

Поэтому приходится использовать другие низкоуровневые бибилиотеки: System.Windows.Forms

### Детектирования клика мыши за пределами окон

using System;

using System.Windows.Forms;

using System.Threading;

Thread mousethread = new Thread(() => SystemEvents.ListenForMouseClicks(e3));

mousethread.Start();

….

class SystemEvents

{

public static void ListenForMouseClicks(e3Application e3)

{

while (true)

{

if (System.Windows.Forms.Control.MouseButtons == MouseButtons.Left)

{

// Фиксируем клик мыши

e3.PutMessage("Левый клик мыши зафиксирован!");

// Здесь можно добавить логику для обработки клика

}

Thread.Sleep(100); // Задержка для уменьшения нагрузки на процессор

}

}

}

### Координаты курсора за пределами окон

using System.Windows.Forms;

System.Windows.Point getpos = new System.Windows.Point(System.Windows.Forms.Cursor.Position.X, System.Windows.Forms.Cursor.Position.Y);

getpos.X

getpos.Y

## \_\_\_\_\_\_\_\_События и другое\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

## События

### OnStartup vs Конструктор App

Конструктор:

App app = new App();

app.InitializeComponent();

public App()

{

// Создаём очередь событий

this.queue = new BlockingCollection<Action>();

// Подписываемся на глобальные события мыши

\_globalHook = Hook.GlobalEvents();

\_globalHook.MouseDown += OnMouseDown;

}

Конструктор вызывается при создании экземпляра класса App (App app = new App()), то есть до запуска приложения.

OnStartup:

protected override void OnStartup(StartupEventArgs e)

{

base.OnStartup(e);

// Создаём очередь событий

this.queue = new BlockingCollection<Action>();

// Подписываемся на глобальные события мыши

\_globalHook = Hook.GlobalEvents();

\_globalHook.MouseDown += OnMouseDown;

}

Метод OnStartup вызывается после App.Run(), но до отображения главного окна .

### Событие на глобальный клик мыши

1. Скачать NuGet пакет MouseKeyHook
2. код

using Gma.System.MouseKeyHook;

private IKeyboardMouseEvents \_globalHook;

public App()

{

// Подписываемся на глобальные события мыши

\_globalHook = Hook.GlobalEvents();

\_globalHook.MouseDown += OnMouseDown;

}

private void OnMouseDown(object sender, System.Windows.Forms.MouseEventArgs e)

{

if (e.Button == System.Windows.Forms.MouseButtons.Left)

{

e3.PutMessage("Левый клик мыши зафиксирован!");

function();

}

}

private void function()

{

……

}

## Создание событий

## Подписывание на событие

this.SizeChanged += MainWindow\_SizeChanged;

## \_\_\_\_\_\_\_\_Управление пространством\_\_\_\_\_\_\_\_\_\_\_\_\_

### Grid

#### Динамическое добавление объектов на grid

grid = new Grid();

grid.ColumnDefinitions.Add(new ColumnDefinition());

this.Content = grid;

newbutton(grid);

newbutton(grid);

newbutton(grid);

newbutton(grid);

}

private void newbutton(Grid grid)

{

grid.RowDefinitions.Add(new RowDefinition());

int rowCount = grid.RowDefinitions.Count;

int columnCount = grid.ColumnDefinitions.Count;

Button button = new Button

{

Content = "Click Me",

HorizontalAlignment = HorizontalAlignment.Center,

VerticalAlignment = VerticalAlignment.Center

};

Grid.SetRow(button, rowCount-1);

Grid.SetColumn(button, columnCount-1);

grid.Children.Add(button);

}

#### Перераспределение объектов в зависимости от ширины окна

public partial class MainWindow : Window

{

Grid grid;

List<Button> buttons = new List<Button>();

public MainWindow()

{

InitializeComponent();

// Создаем новый Grid

grid = new Grid();

grid.ColumnDefinitions.Add(new ColumnDefinition());

this.Content = grid;

this.SizeChanged += MainWindow\_SizeChanged;

for (int i = 1; i < 10; i++)

{

Button btn;

btn = newbutton(grid, i); buttons.Add(btn);

}

}

private Button newbutton(Grid grid, int i)

{

int rowHeight = 50;

grid.RowDefinitions.Add(new RowDefinition { Height = new GridLength(rowHeight) });

int rowCount = grid.RowDefinitions.Count;

int columnCount = grid.ColumnDefinitions.Count;

Button button = new Button

{

Content = "Click Me" + i.ToString(),

HorizontalAlignment = HorizontalAlignment.Center,

VerticalAlignment = VerticalAlignment.Center

};

Grid.SetRow(button, rowCount-1);

Grid.SetColumn(button, columnCount-1);

grid.Children.Add(button);

return button;

}

private void MainWindow\_SizeChanged(object sender, SizeChangedEventArgs e)

{

double windowWidth = e.NewSize.Width;

// Определяем количество столбцов в зависимости от ширины окна

int newColumnCount = (int)(windowWidth / 100); // Например, новый столбец каждые 100 пикселей

if (newColumnCount < 1) newColumnCount = 1; // Минимум 1 столбец

// Если количество столбцов изменилось

if (newColumnCount != grid.ColumnDefinitions.Count)

{

UpdateGridColumns(newColumnCount);

RedistributeButtons();

}

}

private void UpdateGridColumns(int newColumnCount)

{

// Очищаем текущие столбцы

grid.ColumnDefinitions.Clear();

// Добавляем новые столбцы

for (int i = 0; i < newColumnCount; i++)

{

grid.ColumnDefinitions.Add(new ColumnDefinition());

}

}

private void RedistributeButtons()

{

int columnCount = grid.ColumnDefinitions.Count;

int rowCount = (int)Math.Ceiling((double)buttons.Count / columnCount);

// Очищаем текущие строки

grid.RowDefinitions.Clear();

// Добавляем новые строки

for (int i = 0; i < rowCount; i++)

{

grid.RowDefinitions.Add(new RowDefinition { Height = new GridLength(50) });

}

// Перераспределяем кнопки по новым строкам и столбцам

for (int i = 0; i < buttons.Count; i++)

{

int row = i / columnCount;

int column = i % columnCount;

Grid.SetRow(buttons[i], row);

Grid.SetColumn(buttons[i], column);

}

}