# Цель и задачи лабораторной работы

**Цель:** научиться использовать обратные асинхронные вызовы для организации ожидания завершения выполнения асинхронного метода.

**Задачи:**

1. Изучить возможности применения обратных асинхронных вызовов для параллельных потоков;
2. Изучить возможности асинхронного обратного вызова при использовании типа делегата;
3. Изучить возможности асинхронного обратного вызова при использовании лямбда-выражения.

# Реализация индивидуального задания

Согласно варианту задания, требуется:

Создать делегат вида лямбдавыражение

Создать метод, возвращающий результат шифрования строки: каждый исходный символ строки заменяется шифрованным символом, код которого на n больше кода исходного символа с двумя входными параметрами: исходная строка, число сдвига n.

Использовать метод обратного вызова: делегат

## Листинг программного кода

**using** System.Text;

***namespace*** lab4

{

***class*** Program

    {

**public** **static** **async** Task PostTask(Task<**string**> previousTask)

        {

            Console.WriteLine();

            Console.WriteLine("Пост-обработка");

**await** Task.Delay(500);

            Console.WriteLine();

            Console.WriteLine("Работа завершена!");

            Console.WriteLine($"Полученная строка: {**await** previousTask}");

        }

**public** ***delegate*** Task<**string**> CustomDelegate(**string** original, **int** shift);

**public** ***delegate*** Task ContinueWithDelegate(Task<**string**> task);

*// Пример использования с ожиданием через await (основной способ)*

**public** **static** **async** Task Main()

        {

**string** originalStr **=** "Hello, World!";

            Console.WriteLine($"Оригинальная строка: {originalStr}");

**int** shiftStr **=** 3;

            CustomDelegate lambdaDelegate **=** **async** (original, shift) **=>**

            {

                Console.WriteLine("Шифровка началась");

**await** Task.Delay(1000);

*// Task.Run используется для выноса CPU-bound операции в пул потоков*

                StringBuilder encrypted **=** **new**();

*foreach* (**char** c *in* original)

                {

*// Простой алгоритм шифра Цезаря для символов английского алфавита*

*if* (**char**.IsLetter(c))

                    {

**char** baseChar **=** **char**.IsUpper(c) **?** 'A' **:** 'a';

**char** shifted **=** (**char**)(((c **-** baseChar **+** shift) **%** 26) **+** baseChar);

                        encrypted.Append(shifted);

                    }

*else*

                    {

                        encrypted.Append(c); *// Не-буквы оставляем как есть*

                    }

                }

*return* encrypted.ToString();

            };

*// Механизм "обратного вызова" реализован через продолжение кода после await*

**string** encrypted **=** **await** lambdaDelegate(originalStr, shiftStr);

            Console.WriteLine($"Зашифрованная строка: {encrypted}");

            ContinueWithDelegate postDelegate **=** PostTask;

*// Демонстрация использования с ContinueWith (явное продолжение)*

            Task encryptionTask **=** lambdaDelegate(originalStr, shiftStr).ContinueWith(task **=>** postDelegate(task)).Unwrap(); *// Unwrap преобразует вложенную операцию в часть основной*

*// Task encryptionTask = await lambdaDelegate(originalStr, shiftStr).ContinueWith(task => postDelegate(task)); // Переменной encryptionTask присваивается задача, возвращаемая ContinueWith (т. е. PostDelegate)*

*// Task encryptionTask = lambdaDelegate(originalStr, shiftStr).ContinueWith(async task => await postDelegate(task)).Unwrap(); // ContinueWith из-за async task вернёт Task даже до того, как postDelegate закончит работу*

*// А лучше всего делать так:*

*// await PostTask(Task.FromResult(encrypted));*

            Console.WriteLine("Идет шифрование... выполняется другая работа...");

*while* (**!**encryptionTask.IsCompleted)

            {

                Console.Write(". ");

**await** Task.Delay(100);

            }

            Console.WriteLine();

            Console.WriteLine("--Цикл завершён--");

            Console.WriteLine("Теперь всё готово!");

        }

    }

}

## Описание кода

Использование обратных вызовов для организации ожидания завершения асинхронных методов через механизмы ContinueWith и await.

**Ключевые компоненты**

**1. Основной асинхронный метод**

* **Шифрование Цезаря** - сдвиг символов на указанное количество позиций
* **CPU-bound операция** выполняется в пуле потоков через Task.Run (закомментировано)
* **Имитация задержки** - Task.Delay(1000) для асинхронности
* **Метод**ContinueWith - явное указание продолжения
* Unwrap() - преобразует вложенную задачу в плоскую структуру

**2. Мониторинг выполнения**

* **Цикл опроса** состояния задачи через IsCompleted
* **Визуальная индикация** прогресса (точки)
* **Ожидание завершения** всех операций

## Результат работы программы

Оригинальная строка: Hello, World!

Шифровка началась

Зашифрованная строка: Khoor, Zruog!

Шифровка началась

Идет шифрование... выполняется другая работа...

. . . . . . . . . .

Пост-обработка

. . . .

Работа завершена!

Полученная строка: Khoor, Zruog!

--Цикл завершён--

Теперь всё готово!

# Контрольные вопросы

1. **Поясните назначение каждого параметра метода BeginInvoke( ).**
   1. **Сигнатура:** BeginInvoke(params, AsyncCallback, object)
   2. **params** - параметры основного метода делегата
   3. **AsyncCallback** - делегат обратного вызова, выполняемый при завершении асинхронной операции
   4. **object** - пользовательский объект состояния, передаваемый в callback
2. **Почему при использовании типа делегата в качестве метода обратного вызова последний параметр метода BeginInvoke( ) можно не использовать?**
   1. Можно передать null, если не требуется передача состояния
   2. Альтернативно, можно захватить нужные переменные через замыкание
   3. В callback-методе состояние доступно через IAsyncResult.AsyncState, но не всегда необходимо
3. **Опишите области использования асинхронных делегатов. В каких типах проектов .NET Framework они применимы?**
   1. **Типы проектов .NET Framework:**
      1. **Windows Forms** - для сохранения отзывчивости UI
      2. **WPF приложения** - аналогично WinForms
      3. [**ASP.NET**](https://asp.net/)**WebForms** - асинхронные страницы
      4. **Console приложения** - параллельная обработка
      5. **Windows Services** - фоновая обработка
   2. **Основные сценарии применения:**
      1. Длительные вычисления без блокировки UI
      2. Параллельная обработка данных
      3. Вызов веб-сервисов и API
      4. Работа с файловой системой и БД