Лабораторная работа №5

Цель работы:

Познакомиться с многопоточностью. Научиться реализовывать шаблон Event Handler.

Необходимые теоретические сведения

Потоки

Одним из ключевых аспектов в современном программировании является **многопоточность**. Ключевым понятием при работе с многоопоточностью является поток. Поток представляет некоторую часть кода программы. При выполнении программы каждому потоку выделяется определенный квант времени. И при помощи многопоточности мы можем выделить в приложении несколько потоков, которые будут выполнять различные задачи одновременно. Если у нас, допустим, графическое приложение, которое посылает запрос к какому-нибудь серверу или считывает и обрабатывает огромный файл, то без многопоточности у нас бы блокировался графический интерфейс на время выполнения задачи. А благодаря потокам мы можем выделить отправку запроса или любую другую задачу, которая может долго обрабатываться, в отдельный поток. Поэтому, к примеру, клиент-серверные приложения (и не только они) практически не мыслимы без многопоточности.

Основной функционал для использования потоков в приложении сосредоточен в пространстве имен **System.Threading**. В нем определен класс, представляющий отдельный поток - класс **Thread**.

Класс Thread определяет ряд методов и свойств, которые позволяют управлять потоком и получать информацию о нем. Основные свойства класса:

- Статическое свойство **CurrentContext** позволяет получить контекст, в котором выполняется поток
- Статическое свойство **CurrentThread** возвращает ссылку на выполняемый поток
- Свойство **IsAlive** указывает, работает ли поток в текущий момент
- Свойство **IsBackground** указывает, является ли поток фоновым
- Свойство **Name** содержит имя потока
- Свойство **Priority** хранит приоритет потока - значение перечисления ThreadPriority
- Свойство **ThreadState** возвращает состояние потока - одно из значений перечисления ThreadState

Некоторые методы класса Thread:

- Статический метод **GetDomain** возвращает ссылку на домен приложения
- Статический метод **GetDomainID** возвращает id домена приложения, в котором выполняется текущий поток
- Статический метод **Sleep** останавливает поток на определенное количество миллисекунд
- Метод **Abort** уведомляет среду CLR о том, что надо прекратить поток, однако прекращение работы потока происходит не сразу, а только тогда, когда это становится возможно. Для проверки завершенности потока следует опрашивать его свойство ThreadState
- Метод **Interrupt** прерывает поток на некоторое время
- Метод **Join** блокирует выполнение вызвавшего его потока до тех пор, пока не завершится поток, для которого был вызван данный метод
- Метод **Start** запускает поток

Получение информации о потоке

Используем вышеописанные свойства и методы для получения информации о потоке:

```
using System.Threading;
...................
static void Main(string[] args)
{
    // получаем текущий поток
    Thread t = Thread.CurrentThread;

    //получаем имя потока
    Console.WriteLine($"Имя потока: {t.Name}");
    t.Name = "Метод Main";
    Console.WriteLine($"Имя потока: {t.Name}");

    Console.WriteLine($"Запущен ли поток: {t.IsAlive}");
    Console.WriteLine($"Приоритет потока: {t.Priority}");
    Console.WriteLine($"Статус потока: {t.ThreadState}");

    // получаем домен приложения
    Console.WriteLine($"Домен приложения: {Thread.GetDomain().FriendlyName}");

    Console.ReadLine();
}
```

В этом случае мы получим примерно следующий вывод:

```
Имя потока:
Имя потока: Метод Main
Запущен ли поток: True
Приоритет потока: Normal
Статус потока: Running
Домен приложения: HelloApp
```

Так как по умолчанию свойство Name у объектов Thread не установлено, то в первом случае мы получаем в качестве значения этого свойства пустую строку.

Статус потока

Статусы потока содержатся в перечислении **ThreadState**:

- **Aborted**: поток остановлен, но пока еще окончательно не завершен
- **AbortRequested**: для потока вызван метод Abort, но остановка потока еще не произошла
- **Background**: поток выполняется в фоновом режиме
- **Running**: поток запущен и работает (не приостановлен)
- **Stopped**: поток завершен
- **StopRequested**: поток получил запрос на остановку
- **Suspended**: поток приостановлен
- **SuspendRequested**: поток получил запрос на приостановку
- **Unstarted**: поток еще не был запущен
- **WaitSleepJoin**: поток заблокирован в результате действия методов Sleep или Join

В процессе работы потока его статус многократно может измениться под действием методов. Так, в самом начале еще до применения метода Start его статус имеет значение Unstarted. Запустив поток, мы изменим его статус на Running. Вызвав метод Sleep, статус изменится на WaitSleepJoin. А применяя метод Abort, мы тем самым переведем поток в состояние AbortRequested, а затем Aborted, после чего поток окончательно завершится.

Приоритеты потоков

Приоритеты потоков располагаются в перечислении **ThreadPriority**:

- **Lowest**

- **BelowNormal**
- **Normal**
- **AboveNormal**
- **Highest**

По умолчанию потоку задается значение Normal. Однако мы можем изменить приоритет в процессе работы программы. Например, повысить важность потока, установив приоритет Highest. Среда CLR будет считывать и анализировать значения приоритета и на их основании выделять данному потоку то или иное количество времени.

Используя класс Thread, мы можем выделить в приложении несколько потоков, которые будут выполняться одновременно.

Запуск потоков

Во-первых, для запуска нового потока нам надо определить задачу в приложении, которую будет выполнять данный поток. Для этого мы можем добавить новый метод, производящий какие-либо действия.

Для создания нового потока используется делегат **ThreadStart**, который получает в качестве параметра выполняемый метод. И чтобы запустить поток, вызывается метод **Start**.

Рассмотрим на примере:

```
using System.Threading;

class Program
{
    static void Main(string[] args)
    {
        // создаем новый поток
        Thread myThread = new Thread(new ThreadStart(Count));
        myThread.Start(); // запускаем поток

        for (int i = 1; i < 9; i++)
        {
            Console.WriteLine("Главный поток:");
            Console.WriteLine(i * i);
            Thread.Sleep(300);
        }

        Console.ReadLine();
    }

    public static void Count()
    {
        for (int i = 1; i < 9; i++)
        {
            Console.WriteLine("Второй поток:");
            Console.WriteLine(i * i);
            Thread.Sleep(400);
        }
    }
}
```

Здесь новый поток будет производить действия, определенные в методе Count. В данном случае это возведение в квадрат числа и вывод его на экран. И после каждого умножения с помощью метода Thread.Sleep мы усыпляем поток на 400 миллисекунд. Чтобы запустить этот метод в качестве второго потока, мы сначала создаем объект потока: Thread myThread = new Thread(new ThreadStart(Count));. В конструктор передается делегат ThreadStart, который в качестве параметра принимает метод Count. И следующей строкой myThread.Start() мы запускаем поток. После этого управление

передается главному потоку, и выполняются все остальные действия, определенные в методе Main.

Таким образом, в нашей программе будут работать одновременно главный поток, представленный методом Main, и второй поток. Кроме действий по созданию второго потока, в главном потоке также производятся некоторые вычисления. Как только все потоки отработают, программа завершит свое выполнение.

Подобным образом мы можем создать и три, и четыре, и целый набор новых потоков, которые смогут решать те или иные задачи.

Существует еще одна форма создания потока: Thread myThread = new Thread(Count); Хотя в данном случае явным образом мы не используем делегат ThreadStart, но неявно он создается. Компилятор C# выводит делегат из сигнатуры метода Count и вызывает соответствующий конструктор.

Timer

Одним из важнейших классов, находящихся в пространстве имени System.Threading, является класс **Timer**. Данный класс позволяет запускать определенные действия по истечению некоторого периода времени.

Например, нам надо запускать какой-нибудь метод через каждые 2000 миллисекунд, то есть раз в две секунды:

```
class Program
{
    static void Main(string[] args)
    {
        int num = 0;
        // устанавливаем метод обратного вызова
        TimerCallback tm = new TimerCallback(Count);
        // создаем таймер
        Timer timer = new Timer(tm, num, 0, 2000);

        Console.ReadLine();
    }
    public static void Count(object obj)
    {
        int x = (int)obj;
        for (int i = 1; i < 9; i++, x++)
        {
            Console.WriteLine($"{x * i}");
        }
    }
}
```

Первым делом создается объект делегата **TimerCallback**, который в качестве параметра принимает метод. Причем данный метод должен в качестве параметра принимать объект типа object.

И затем создается таймер. Данная перегрузка конструктора таймера принимает четыре параметра:

- объект делегата TimerCallback
- объект, передаваемый в качестве параметра в метод Count
- количество миллисекунд, через которое таймер будет запускаться. В данном случае таймер будет запускать немедленно после создания, так как в качестве значения используется 0
- интервал между вызовами метода Count

И, таким образом, после запуска программы каждые две секунды будет срабатывать метод Count.

Если бы нам не надо было бы использовать параметр obj у метода Count, то при создании таймера мы могли бы указывать в качестве соответствующего параметра знаечние null: Timer timer = new Timer(tm, null, 0, 2000);

Задание

Написать C# программу на основе лабораторной №1,2,3,4 реализующую работу с классами по вариантам.

1. Создать CRUD(CReate, Update, Delete) интерфейс для управления классами каждого типа.
2. Создать контроллер, наследованный от CRUD интерфейса, который в отдельном потоке должен писать изменённые данные в файл.
3. Создать интерфейс IEventHandler с методами OnInsert, OnUpdate, OnDelete.
4. Создать интерфейс IDataAccessor, позволяющий запрашивать список GUID и элемент с заданным GUID.
5. Класс-хранилище объектов четырёх типов должен наследовать интерфейс IDataAccessor.
6. Класс-хранилище объектов четырёх типов должен иметь возможность регистрировать у себя объекты наследники интерфейса IEventHandler (слушатели).
7. Класс-хранилище объектов четырёх типов по таймеру должен вычитывать данные из файла, при изменении данных вызывать соответствующий метод для всех зарегистрированных слушателей.

Разработанная программа должна быть консольной, позволяющей на выбор добавлять/изменять/удалять объекты классов, при наличии точно такого же экземпляра сообщать об этом пользователю. После совершения действия должно выводится подтверждение о его выполнении. Кроме того должна быть возможность просмотреть все объекты заданного класса.

В отчёт обязательно построить диаграмму классов, описывающую иерархию созданных классов.

Варианты:

1. Студент, преподаватель, персона, заведующий кафедрой
2. Служащий, персона, рабочий, инженер
3. Рабочий, кадры, инженер, администрация
4. Деталь, механизм, изделие, узел
5. Организация, страховая компания, нефтегазовая компания, завод
6. Журнал, книга, печатное издание, учебник
7. Тест, экзамен, выпускной экзамен, испытание
8. Место, область, город, мегаполис
9. Игрушка, продукт, товар, молочный продукт
10. Квитанция, накладная, документ, счет
11. Автомобиль, поезд, транспортное средство, экспресс
12. Двигатель, двигатель внутреннего сгорания, дизель, реактивный двигатель
13. Республика, монархия, королевство, государство
14. Млекопитающее, парнокопытное, птица, животное
15. Корабль, пароход, парусник, корвет