МИНИСТЕРСТВО ПРОСВЕЩЕНИЯ
РОССИЙСКОЙ ФЕДЕРАЦИИ

ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ
БЮДЖЕТНОЕ ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ
ВЫСШЕГО ОБРАЗОВАНИЯ
«ЛУГАНСКИЙ ГОСУДАРСТВЕННЫЙ
ПЕДАГОГИЧЕСКИЙ УНИВЕРСИТЕТ»

В. В. Швыров
Д. А. Капустин
В. Н. Шишлакова

ПРОГРАММИРОВАНИЕ ДЛЯ ПЛАТФОРМЫ .NET

Часть 3. РАБОТА С БАЗАМИ ДАННЫХ И МНОГОЗАДАЧНОЕ ПРОГРАММИРОВАНИЕ

Лабораторный практикум
для студентов 3 курса очной и заочной форм обучения
направления подготовки
44.03.04 Профессиональное обучение (по отраслям),
профиль: «Моделирование цифровых платформ»

Луганск
Издательство ЛГПУ
2024

УДК [004.04:004.514] (076.5)
ББК 32.973.202-018.2р3
Ш 35

Рецензенты:

Кривко Я. П. заведующий кафедрой высшей математики и методики преподавания математики федерального государственного бюджетного образовательного учреждения высшего образования «Луганский государственный педагогический университет», доктор педагогических наук, доцент;

Мальцев Я. И. доцент кафедры прикладной математики федерального государственного бюджетного образовательного учреждения высшего образования «Луганский государственный университет имени Владимира Даля», кандидат технических наук, доцент;

Давыскиба О. В. доцент кафедры фундаментальной математики федерального государственного бюджетного образовательного учреждения высшего образования «Луганский государственный педагогический университет», кандидат педагогических наук, доцент.

Швыров, В. В., Капустин, Д. А., Шишлакова, В. Н.

Ш35 Программирование для платформы .Net. Часть 3. Работа с базами данных и многозадачное программирование: лабораторный практикум / В. В. Швыров, Д. А. Капустин, В. Н. Шишлакова; ФГБОУ ВО «ЛГПУ». – Луганск : Издательство ЛГПУ, 2024. – 184 с.

«Часть 3. Работа с базами данных и многозадачное программирование» лабораторного практикума «Программирование для платформы .NET» содержит заключительный блок из 12 лабораторных работ. Особое внимание уделяется асинхронному и параллельному программированию.

Предназначен для студентов 3 курса очной и заочной форм обучения направления подготовки 44.03.04 Профессиональное обучение (по отраслям), профиль: «Моделирование цифровых платформ».

УДК [004.65:004.514] (076.5)
ББК 32.973.202-018.2р3

Рекомендовано Учебно-методическим советом ФГБОУ ВО «ЛГПУ» в качестве лабораторного практикума для студентов очной и заочной форм обучения, обучающихся по направлению подготовки 44.03.04 Профессиональное обучение (по отраслям). Моделирование цифровых платформ (протокол № 10 от 21 мая 2024)

Содержание

Введение

Данный лабораторный практикум предназначен для выполнения лабораторных работ по дисциплине «Программирование для платформы .Net» для студентов очной и заочной форм обучения по направлению подготовки 44.03.04 «Профессиональное обучение (по отраслям)», профиль: «Моделирование цифровых платформ». В соответствии с учебным планом, по данной дисциплине предусмотрено 38 лабораторных работ. В третьей части лабораторного практикума представлен заключительный блок из 12 лабораторных работ. Особое внимание уделяется следующим темам: асинхронное и параллельное программирование. Студенты получают возможность понять и научиться применять асинхронные и параллельные методики программирования. Эти навыки становятся все более важными в современной разработке программного обеспечения, где эффективное использование ресурсов и обработка множества задач одновременно играют важную роль.

Также детально изучается работа с базами данных, валидация моделей, работа с криптографическими примитивами. В частности, студенты изучают основы создания, запросов и управления базами данных, что является фундаментальным навыком для разработчиков приложений. В условиях повышенной ценности данных и конфиденциальности информации, знание криптографии становится критически важным. Студенты изучают принципы криптографии и ее применение в защите данных. Работа с датой и временем часто является неотъемлемой частью разработки приложений.

В структуре каждой лабораторной работы содержится блок, в котором представлены краткие теоретические сведения по теме работы, а также непосредственно задания для выполнения и контрольные вопросы.

Для успешной защиты лабораторной работы студенту необходимо детально изучить представленные теоретические сведения, выполнить все поставленные задачи, уметь ответить на

контрольные вопросы. Кроме того, преподавателю необходимо предоставить все исходные файлы, которые содержат решения задач, а также сформировать отчет о выполненной работе.

Отчет по лабораторной работе должен содержать:

– титульную страницу, на которой указаны название ВУЗа, института, кафедры, тема лабораторной работы, курс, группа, ФИО студента, и другие сведения в соответствии с установленном образцом;

– основную часть, в которой приводится решение всех поставленных задач в виде последовательности скриншотов IDE, которые подписываются строкой комментарием, с кратким описанием действий;

– все листинги исходных кодов разработанных приложений;

– ответы на контрольные вопросы к лабораторной работе;

– выводы о том, что было сделано в работе.

Лабораторная работа № 27

Тема: **«Многопоточное программирование в C#».**

Цель: изучить принципы и основы многопоточного программирования на языке C# для создания эффективных и стабильных приложений, способных обрабатывать множество задач одновременно.

Краткие теоретические сведения

Одним из ключевых моментов программирования является многопоточность. Поток – это некий путь выполнения кода. Даже если мы создаем обычное консольное приложение типа «Hello world», то создается, как минимум один (он же главный) поток, который начинает свой путь с метода Main. Каждому потоку в приложении выделяется определенное количество времени (квант), в течение которого поток выполняет возложенную задачу. И, даже, если компьютер оснащен одноядерным процессором, можно создавать несколько потоков в приложении, однако это будет, скорее, псевдо-многопоточность в рамках которой потоки просто будут очень быстро переключаться между собой, обеспечивая тем самым эффект многопоточности.

Вариантов, когда можно использовать многопоточную модель в приложении можно привести несколько. Например, самый очевидный – ожидание ответа от какого-либо сервера в сети. Если использовать один поток, в рамках которого приложение будет ожидать ответ от сервера, а затем выполнять другие задачи, то, если ответ будет идти достаточно долго, программа будет «притормаживать». Намного хуже будет в том случае, если сервер вообще не ответит – в этом случае программа просто «зависнет» и не двинется дальше до полной перезагрузки. В этом случае имеет смысл вынести работу с сервером в отдельный поток.

Другой пример – работа MS Word. Как известно, Word при наборе текста автоматически проверяет правописание и, в случае нахождения ошибок подчеркивает их. Если бы в Word использовалась однопоточная модель, то вряд ли этот текстовый редактор имел бы большую популярность. Проверка правописания

и прочие «фишки» Word выполняются в отдельных потоках, не нарушая и не притормаживая при этом нашу непосредственную работу в редакторе.

1. Класс Thread

Основные типы и классы для работы с многопоточностью в C# располагаются в пространстве имен System.Threading и главным из классов является класс Thread. Именно благодаря Thread можно создавать новые потоки в нашем приложении и делегировать им выполнение каких-либо задач, а использование статических методов этого класса позволяет управлять уже существующими потоками приложения.

Свойства класса Thread

Ниже представлены основные свойства класса Thread.

CurrentThread – возвращает выполняющийся в данный момент поток.

ExecutionContext – возвращает объект ExecutionContext, содержащий сведения о различных контекстах текущего потока.

IsAlive – возвращает значение, показывающее статус выполнения текущего потока.

IsBackground – возвращает или задает значение, показывающее, является ли поток фоновым.

IsThreadPoolThread – возвращает значение, показывающее, принадлежит ли поток к группе управляемых потоков (пулу потоков).

ManagedThreadId – возвращает уникальный идентификатор текущего управляемого потока.

Name – получает или задает имя потока.

Priority – возвращает или задает значение, указывающее на планируемый приоритет потока.

ThreadState – возвращает значение, содержащее состояние текущего потока.

Посмотрим, какие значения будут содержать эти свойства для главного потока приложения типа «Hello world»:

```
using System;
using System.Threading;
namespace Multithread
```

```
{
internal class Program
{
static void Main(string[] args)
{
Thread thread = Thread.CurrentThread;
Console.WriteLine($"Name {thread.Name}");
thread.Name = "Главный поток приложения";
Console.WriteLine($"Name {thread.Name}");
Console.WriteLine($"ManagedThreadId
{thread.ManagedThreadId}");
Console.WriteLine($"ThreadState {thread.ThreadState}");
Console.WriteLine($"IsAlive {thread.IsAlive}");
Console.WriteLine($"IsBackground {thread.IsBackground}");
Console.WriteLine($"IsThreadPoolThread
{thread.IsThreadPoolThread}");
Console.WriteLine($"Priority {thread.Priority}");
}}}
```

Результат выполнения:

```
Name
Name Главный поток приложения
ManagedThreadId 1
ThreadState Running
IsAlive True
IsBackground False
IsThreadPoolThread False
Priority Normal
```

Как можно видеть по итогу работы программы, в начале главный поток не имеет никакого имени, однако, можно его назначить, используя свойство Name.

Состояние потока может многократно меняться в процессе выполнения приложения. По умолчанию состояние потока Unstarted, после запуска потока методом Start его статус изменяется на Running, вызвав метод Sleep() мы, тем самым, переведем поток в статус WaitSleepJoin и т.д. По умолчанию, потоку устанавливается приоритет Normal, однако, можно изменить его и назначить другое значение.

Рассмотрим приложение, демонстрирующее работу двух потоков:

```
using System;
using System.Threading;
namespace Multithread
```

```
{
internal class Program
{
static void Main(string[] args)
{
Thread thread = new Thread(new ThreadStart(Proc));
thread.Start();
string s;
do
{
s = Console.ReadLine();
Console.WriteLine(s);
}
while (s != "q");
}
public static void Proc()
{
for (int i = 0; i < 10; i++)
{
Console.WriteLine($"Это {i} итерация цикла в потоке");
Thread.Sleep(1000);
}}}}
```

Выполнение приложения начинается с метода `Main` в котором создаем и запускаем поток thread. Этот поток 10 раз выводит в консоль сообщение. Главный же поток работает до тех пор, пока мы не введем в консоль символ `q` и не нажмем Enter. Это пример, не имеющий какой-либо практической ценности, однако он прекрасно демонстрирует, во-первых, работу двух потоков в приложении (главного и вторичного), а, во-вторых, как приложение продолжает работу даже в том случае, если главный поток прекращает свою работу. Попробуйте запустить приложение, сразу написать `q` и нажать Enter – после этого вы уже не сможете ничего писать в консоль, так как главный поток прекратит работу, однако вторичный (созданный нами) поток продолжит работу и приложение завершит свою работу только после того, как вторичный поток отработает все 10 итераций цикла.

Секрет такого поведения кроется в свойстве `IsBackground`. Дело в том, что все потоки по умолчанию создаются с приоритетом `Normal` и значением `IsBackground` равным `false`, т.е. как потоки

переднего плана. И до тех пор, пока в приложении есть хотя бы один поток переднего плана, не завершивший задачу, приложение будет работать. Попробуем создать наш поток как фоновый:

```
internal class Program
{
static void Main(string[] args)
{
Thread thread = new Thread(new ThreadStart(Proc));
thread.IsBackground = true;//поток теперь фоновый
thread.Start();
string s;
do
{
s = Console.ReadLine();
Console.WriteLine(s);
}
while (s != "q");
}
public static void Proc()
{
for (int i = 0; i < 10; i++)
{
Console.WriteLine($"Это {i} итерация цикла в потоке");
Thread.Sleep(1000);
}}}
```

Теперь, если запустить приложение и набрать `q`, то приложение сразу же прекратит свою работу вне зависимости от того прошел ли вторичный поток `10` итераций цикла или нет.

2. Делегаты потоков. Делегат `ThreadStart`

Ранее мы использовали вот такой конструктор создания потока:

```
Thread thread = new Thread(new ThreadStart(Proc));
```

`ThreadStart` – это ни что иное, как делегат, который сообщает нам, что метод не должен ничего принимать и ничего возвращать. Например, можно написать вот такое приложение, используя для потока делегат `ThreadStart`:

```
internal class Program
{
static void Main(string[] args)
{
Thread thread = new Thread(new ThreadStart(Proc));
thread.Start();
```

```
for (int i = 0; i < 10; i++)
{
Console.WriteLine($"Значение из главного потока: {i * i * i}");
Thread.Sleep(500);
}
}
public static void Proc()
{
for (int i = 0; i < 10; i++)
{
Console.WriteLine($"Значение из второго потока: {i*i}");
Thread.Sleep(1000);
}}}
```

Второй поток будет выполнять метод Proc, который мы передали в качестве параметра делегату ThreadProc. Так как вторичный поток не определен как фоновый (свойство IsBackground по умолчанию равно false), то в консоли увидим примерно следующие результаты:

```
Значение из главного потока: 0
Значение из второго потока: 0
Значение из главного потока: 1
Значение из второго потока: 1
Значение из главного потока: 8
```

3. Делегат ParameterizedThreadStart

Если нам необходимо передать какие-либо параметры в поток, например, начальные значения переменных, то можно воспользоваться вторым делегатом – ParameterizedThreadStart. Выглядит этот делегат следующим образом:

```
public delegate void ParameterizedThreadStart(object? obj);
```

то есть, метод, передаваемый в параметре делегата не должен ничего возвращать, но может принимать один параметр типа object. А так как мы помним, что все в C# так или иначе является объектами, то подобный подход позволяет нам передать в поток вообще всё, что угодно – от простого числа до сложных объектов. В качестве демонстрации использования этого делегата, воспользуемся алгоритмом поиска простых чисел и напишем приложение, в котором вторичный поток будет определять простые числа, а главный – выполнять ИБД (Имитацию Бурной Деятельности):

```
using System;
```

```
using System.Threading;
namespace Multithread
{
internal class Program
{
static void Main(string[] args)
{
int N = 10000;
Thread thread = new Thread(new ParameterizedThreadStart(Proc));
thread.Start(N);
for (int i = 0; i < 10; i++)
{
Console.WriteLine($"Имитируем бурную деятельность главного
потока");
Thread.Sleep(1000);
}
}
public static bool IsPrime(int number)
{
for (int i = 2; i < number; i++)
{
if (number % i == 0)
return false;
}
return true;
}
public static void Proc(object N)
{
int n = (int)N;
for (int i = 1; i <= n; i++)
{
if (IsPrime(i))
{
Console.WriteLine($"{i} - простое число");
Thread.Sleep(50);
}}}}}
```

Стоит обратить внимание на то, как создается и запускается вторичный поток. В качестве параметра в конструкторе `Thread` мы используем делегат `ParameterizedThreadStart` в который передаем метод `Proc`. Метод `Proc`, как и требуется, принимает всего один параметр типа `object`, поэтому приводим параметр `N` к нужному нам типу – `int`. Чтобы указать конкретное значение, которое мы хотим передать в поток, используем перегруженную

версию метода Start и в его параметре передаем необходимое значение в поток (в нашем случае – это значение 10000).

Конечно, такая работа с потоками не безопасна, как минимум по причине того, что можно было бы передать в метод Start не число, а например, какой-то сложный объект или строку и тогда мы бы получили исключение. Избежать этого можно относительно просто – объявить специальный класс с необходимым нам методом без параметров и воспользоваться делегатом ThreadStart. Перепишем наше приложение следующим образом:

```
using System;
using System.Threading;
namespace Multithread
{
public class PrimeCounter
{
public int N { get; set; }
private bool IsPrime(int number)
{
for (int i = 2; i < number; i++)
{
if (number % i == 0)
return false;
}
return true;
}
public void Calculate()
{
for (int i = 1; i <= N; i++)
{
if (IsPrime(i))
{
Console.WriteLine($"{i} - простое число");
Thread.Sleep(50);
}}}}
internal class Program
{
static void Main(string[] args)
{
PrimeCounter primeCounter = new PrimeCounter();
primeCounter.N = 10000;
Thread thread = new Thread(new
ThreadStart(primeCounter.Calculate));
thread.Start();
```

```
for (int i = 0; i < 10; i++)
{
Console.WriteLine($"Имируем бурную деятельность главного
потока");
Thread.Sleep(1000);
}}}}
```

Обратите внимание на то, что все необходимые методы, в том числе и метод `IsPrime` вынесены в отдельный класс. Теперь наш код стал типобезопасным – мы уже не сможем передать классу в качестве значения свойства `N` строку. В потоке же воспользовались делегатом `ThreadStart` и передали ему метод `Calculate` класса `PrimeCounter`. При этом, результат работы программы никак не изменился.

Таким образом, нами рассмотренны два делегата, используемых при создании потоков в `C#` – `ThreadStart` и `ParameterizedThreadStart`. Использование делегата `ParameterizedThreadStart` хоть и позволяет передавать в поток какую-либо информацию, однако использование этого делегата может быть сопряжено с проблемами безопасности, так как в поток может передаваться любое значение типа `object`. Более безопасно, с этой точки зрения, оформить все необходимые для потока операции в виде отдельного класса, в котором определить метод без параметров и передать его в качестве параметра для делегата `ThreadStart` в конструкторе потока `Thread`.

4. Конкурентный доступ и синхронизация

Часто различные потоки в приложении используют разделяемые (общие для всех) ресурсы. Например, несколько потоков могут использовать один и тот же файл для записи лога или несколько потоков используют список `List` для чтения/записи элементов и так далее. Как только два и более потоков обращаются к одному и тому же ресурсу приложения, у них возникает конкуренция и предугадать в какой последовательности сработают потоки невозможно. Итог работы будет зависеть от самой операционной системы и того, как система будет выделять ресурсы потокам. Даже вызов метода `Start` и `Thread` не дает

гарантии того, что поток моментально запустится до следующей за Start строки кода.

Рассмотрим пример работы нескольких потоков:

```
internal class Program
{
static int x;
static void Main(string[] args)
{
for (int i = 1; i < 6; i++)
{
Thread thread = new Thread(new ThreadStart(Func));
thread.Name = i.ToString();
thread.Start();
}}
public static void Func()
{
x = 0;
for (int i = 0; i < 10; i++)
{
x++;
Console.WriteLine($"Поток {Thread.CurrentThread.Name} вывел
число {x}");
Thread.Sleep(100);
}}}
```

В этом примере в методе Main создаются пять потоков, каждый из которых получает собственное имя и, затем, выполняет метод Func, в котором обращается к общей для всех потоков переменной x, увеличивая её значение на 1. По сути, ожидаем того, что каждый поток от 1 до 5 выведет в консоль значение x от 1 до 10. По факту же – не можем угадать в какой последовательности будут выводиться значения x, так как очередность выполнения функции потоками определяет за нас операционная система. Чтобы наши потоки работали так, как мы того ожидаем, можно синхронизировать их работу.

Оператор lock имеет следующий синтаксис:

```
lock (x)
{
// Your code...
}
```

Основная идея использования оператора lock заключается в том, что блок кода заключенный в этот оператор в один момент времени может выполняться только одним потоком (т.н. идея

критической секции). Остальные потоки будут ожидать пока текущий поток не выполнит код, и блокировка не будет снята. При этом, в качестве x у оператора lock может выступать только объект ссылочного типа.

Попробуем переписать наш пример следующим образом:

```
internal class Program
{
static int x = 0;
static object locker = new object();
static void Main(string[] args)
{
for (int i = 1; i < 6; i++)
{
Thread thread = new Thread(new ThreadStart(Func));
thread.Name = i.ToString();
thread.Start();
}
}
public static void Func()
{
lock (locker)
{
x = 0;
for (int i = 0; i < 10; i++)
{
x++;
Console.WriteLine($"Поток {Thread.CurrentThread.Name} вывел
число {x}");
Thread.Sleep(100);
}}}}}
```

Теперь, если запустить приложение, то увидим, что каждый поток последовательно выводит значения x от 1 до 10. Следует отметить следующий момент:

Несмотря на то, что в качестве объекта-заглушки у оператора lock может выступать любой объект ссылочного типа, не стоит использовать в операторе ключевое слово this.

То есть, в нашем случае, вот такая конструкция lock сработала бы без проблем (при условии, что работаем с не статическими методами):

```
lock (this)
{
<--тут выполнение кода-->
}
```

Но таким же образом (с использованием this) могут одновременно происходить блокировки и в других частях программы, что может привести к взаимному блокированию нескольких потоков и программа «зависнет». Поэтому лучше потратить несколько секунд времени и одну строку кода и определить для оператора lock свой объект-заглушку.

В целом, идея использования lock показана в примере выше –запрещаем всем прочим потокам работать с общим ресурсом до тех пор, пока текущий поток не выйдет из критической секции. В связи с этим, может возникнуть резонный вопрос – какой смысл использовать критические секции, если они тормозят работу всех потоков? Если одна задача выполняется 1 секунду, то 10 таких задач в 10 потоках будут выполняться также за 1 секунду, а используя lock мы, фактически, возвращаемся к однопоточной схеме и ждем выполнение всех потоков 10 секунд. На первый взгляд может показаться именно так, но не все так просто.

Во-первых, использование lock (критических секций) дает нам гарантию того, что состояние общего ресурса не будет нарушено и потоки отработают так, как мы того от них ожидаем. Например, один поток пишет в список имена файлов из директории, а второй – выводит этот список на экран. Если потоки не синхронизировать, то может случится ситуация, когда второй поток выведет не полный список файлов, т.е. выдаст недостоверную информацию, в то время как первый поток будет продолжать работать, но уже вхолостую (второй-то уже всю свою работу выполнил, и список на экран вывел).

Во-вторых, можно «завернуть» в lock не весь код метода, а только его критическую часть, т.е. ту, в которой используется разделяемый ресурс. В этом случае, потери времени, связанные с синхронизацией, будут меньше и те же 10 однотипных задач могут выполняться не 10, а, скажем, 5 и 3 секунды. Например, каждый поток скачивает из сети файлы различного объема и увеличивает счётчик полученных файлов на 1 при каждой удачной загрузке и выводит имя файла на экран. Загрузку файла из сети можно не блокировать – у каждого потока свой файл, а вот наращивание счётчика на 1 и вывод на экран можно «завернуть» в lock.

5. Способы синхронизации потоков. Монитор (Monitor)

По сути, оператор lock, о котором шла речь в предыдущей части, это т.н. «синтаксический сахар», удобная обертка над классом Monitor. Вот так выглядел наш код с использованием оператора lock:

```
internal class Program
{
static int x = 0;
static object locker = new object();
static void Main(string[] args)
{
for (int i = 1; i < 6; i++)
{
Thread thread = new Thread(new ThreadStart(Func));
thread.Name = i.ToString();
thread.Start();
}}
public static void Func()
{
lock (locker)
{
x = 0;
for (int i = 0; i < 10; i++)
{
x++;
Console.WriteLine($"Поток {Thread.CurrentThread.Name} вывел
число {x}");
Thread.Sleep(100);
}}}}}
```

С использованием Monitor, этот же код будет выглядеть следующим образом:

```
internal class Program
{
static int x = 0;
static object locker = new object();
static void Main(string[] args)
{
for (int i = 1; i < 6; i++)
{
Thread thread = new Thread(new ThreadStart(Func));
thread.Name = i.ToString();
thread.Start();
}}
public static void Func()
{
```

```
bool lockWasTaken = false;
Monitor.Enter(locker, ref lockWasTaken);
try
{
x = 0;
for (int i = 0; i < 10; i++)
{
x++;
Console.WriteLine($"Поток {Thread.CurrentThread.Name} вывел
число {x}");
Thread.Sleep(100);
}}
finally
{
if (lockWasTaken)
Monitor.Exit(locker);
}}}}
```

Метод Montor.Enter принимает два параметра:

1. Объект блокировки (locker)

2. Значение типа bool, указывающее на результат блокировки (true означает, что блокировка успешно выполнена).

Вне зависимости от того, будут ли какие-либо исключительные ситуации (ошибки) в коде, в разделе finally блокировка снимается с использованием метода Exit и следующий поток получает доступ к критической секции для выполнения своей задачи.

Кроме методов Enter/Exit класс Monitor также содержит ряд полезных методов для синхронизации потоков.

IsEntered – определяет, содержит ли текущий поток блокировку указанного объекта

TryEnter – пытается получить эксклюзивную блокировку указанного объекта

Wait – освобождает блокировку объекта и блокирует текущий поток до тех пор, пока тот не получит блокировку снова.

Pulse – уведомляет поток в очереди готовности об изменении состояния объекта с блокировкой

PulseAll – уведомляет все ожидающие потоки об изменении состояния объекта

AutoResetEvent – это класс, который служит целям синхронизации потоков. Представляет событие синхронизации

потоков, которое при срабатывании автоматически сбрасывается, освобождая один поток в состоянии ожидания. Попробуем переписать наш пример синхронизации потоков с использованием класса AutoResetEvent:

```
internal class Program
{
static int x = 0;
static AutoResetEvent ResetEvent = new AutoResetEvent(true);
static object locker = new object();
static void Main(string[] args)
{
for (int i = 1; i < 6; i++)
{
Thread thread = new Thread(new ThreadStart(Func));
thread.Name = i.ToString();
thread.Start();
}}
public static void Func()
{
ResetEvent.WaitOne();
x = 0;
for (int i = 0; i < 10; i++)
{
x++;
Console.WriteLine($"Поток {Thread.CurrentThread.Name} вывел
число {x}");
Thread.Sleep(100);
}
ResetEvent.Set();
}}
```

Разберем этот пример более подробно. Объект ResetEvent может находится в двух состояниях – сигнальном и несигнальном. По умолчанию создаем объект в сигнальном состоянии (new AutoResetEvent(true)).

Все потоки, которые мы создаем, стоят в очереди за этим объектом и получить его они могут только в том случае, если объект находится в сигнальном состоянии. Таким образом, раз создаем этот объект в уже сигнальном состоянии, то первый же созданный нами поток «ловит» этот сигнал и вызывает метод ResetEvent.WaitOne(), что означает, что наш объект ResetEvent переходит в несигнальное состояние – все потоки за текущим потоком снова встают в очередь и ждут, пока текущий поток не

выполнит свою задачу и не «передаст сигнал», используя метод ResetEvent.Set(). И так происходит до тех пор, пока не пройдет вся очередь потоков.

Таким образом, класс AutoResetEvent позволяет управлять синхронизацией потоков, используя «сигналы» и его работа схожа с работой монитора или оператора lock.

6. Мьютекс (Mutex)

Ещё один способ синхронизации потоков – использование мьютексов. В .NET работа с Mutex в чем-то схожа с предыдущим способом синхронизации потоков. Так, наш пример можно переписать с использованием мьютекса следующим образом:

```
internal class Program
{
static int x = 0;
static Mutex mutex = new Mutex();
static object locker = new object();
static void Main(string[] args)
{
for (int i = 1; i < 6; i++)
{
Thread thread = new Thread(new ThreadStart(Func));
thread.Name = i.ToString();
thread.Start();
}}
public static void Func()
{
mutex.WaitOne();
x = 0;
for (int i = 0; i < 10; i++)
{
x++;
Console.WriteLine($"Поток {Thread.CurrentThread.Name} вывел
число {x}");
Thread.Sleep(100);
}
mutex.ReleaseMutex();
}}
```

При использовании мьютекса основную работу по синхронизации потоков выполняют методы WaitOne() и ReleaseMutex().

Метод WaitOne()приостанавливает выполнение потока до тех пор, пока не будет получен объект мьютекса mutex. После выполнения всех необходимых действий мьютекс становится не нужен и поток освобождает его с помощью метода ReleaseMutex().Таким образом, как только поток дойдет до вызова WaitOne(), он будет ожидать освобождения мьютекса и после его получения продолжит выполнять свою задачу.

7. Семафор (Semaphore).

Ещё один способ с говорящим названием для синхронизации потоков – использование семафоров. Семафор позволяет запускать в работу определенное количество потоков, а остальные – приостанавливать, т.е. действует примерно также, как и обычный светофор на дорогах. Например, есть какой-то ночной клуб, который может вмещать одновременно некоторое количество посетителей. Как только достигается заданная емкость (количество посетителей в клубе), то перед клубом вырастает очередь и на каждого вышедшего из клуба посетителя получаем одного входящего. Таким образом, семафор в данном примере будет у нас как вышибала в том самом клубе. Попробуем перевести этот пример на язык программирования:

```
public class Visitor
{
static Semaphore _semaphore = new Semaphore(3, 3);
Thread myThread;
// int _count = 1; //счётчик посещений. Один раз сходить в клуб
- достаточно
public Visitor(int i)
{
myThread = new Thread(Fun);
myThread.Name = $"Посетитель #{i}";
myThread.Start();
}
public void Fun()
{
_semaphore.WaitOne();
Console.WriteLine($"{Thread.CurrentThread.Name}    входит    в
клуб");
Console.WriteLine($"{Thread.CurrentThread.Name} веселится");
Thread.Sleep(1000); //безудержное веселье
```

```
Console.WriteLine($"{Thread.CurrentThread.Name}        покидает
клуб");
_semaphore.Release();
}}
internal class Program
{
static void Main(string[] args)
{
for (int i = 1; i < 7; i++)
{
Visitor visitor = new Visitor(i);
}}}
```

Вся функциональность сосредоточена в классе `Visitor`. Так как семафор (по примеру – это наш вышибала в клубе) должны видеть все посетители, то поле `_semaphore` объявлено как `static`. Можно было бы объявить это поле и в методе `Main`, но мы решили унести его в сам класс для того, чтобы он (класс) был самодостаточным. Семафор создается с двумя параметрами (`new Semaphore(3, 3)`) – первый параметр показывает количество свободных мест (сколько посетителей могут войти в клуб), а второй – общую емкость. Таким образом, изначально считаем, что клуб полностью свободен и три человека могут в него зайти.

Далее, в методе `Fun` наши посетители захватывают семафор (`_semaphore.WaitOne()`), веселятся (`Thread.Sleep(1000)`) и затем покидают клуб, освобождая тем самым семафор (`_semaphore.Release()`). После чего в клуб заходит новый посетитель и т.д.

В методе `Main` нам остается только создать некоторое количество пользователей и наблюдать со стороны за их безудержным весельем.

Стоит отметить, что не всегда посетители (потоки) будут запускаться в том порядке, в котором они созданы. То, в какой последовательности будут стартовать потоки – решает операционная система.

Задания и порядок выполнения работы

Задание 1. Выполните все примеры, представленные в теоретической части. При выводе данных в консоль первой строкой необходимо выводить свое ФИО.

Задание 2. Напишите программу, в которой в главном потоке целочисленная переменная через определенные промежутки получает случайное значение. Два дочерних потока периодически (через определенные промежутки времени) проверяют значение переменной. Первый поток проверяет, является ли значение переменной нечетным, а второй поток проверяет, делится ли значение переменной на 3. Если проверка успешная, то соответствующий поток выводит в консольное окно сообщение.

Задание 3. Создайте программу, которая использует два потока для вычисления суммы элементов массива. Один поток должен вычислять сумму четных элементов, а второй – сумму нечетных элементов.

Задание 4. Создайте приложение, которое использует потоки для параллельной загрузки нескольких файлов из интернета. Отслеживайте прогресс загрузки и выводите информацию о времени, затраченном на каждую загрузку. В качестве примера решения можно модифицировать следующий код:

```
using System;
using System.Collections.Generic;
using System.Net.Http;
using System.Threading.Tasks;

class Program
{
    static async Task Main(string[] args)
    {
        // Список URL-адресов файлов, которые вы хотите
загрузить параллельно
        List<string> fileUrls = new List<string>
        {
            "https://example.com/file1.txt",
            "https://example.com/file2.txt",
            "https://example.com/file3.txt"
        };

        // Создаем HttpClient для выполнения HTTP-запросов
        using (HttpClient httpClient = new HttpClient())
        {
```

```
            // Создаем список задач для загрузки файлов
            List<Task> downloadTasks = new List<Task>();

            foreach (string url in fileUrls)
            {
                // Добавляем задачу для каждого URL-адреса

downloadTasks.Add(DownloadFileAsync(httpClient, url));
            }

            // Дождемся завершения всех задач
            await Task.WhenAll(downloadTasks);

            Console.WriteLine("Все файлы загружены.");
        }
    }

    static async Task DownloadFileAsync(HttpClient httpClient,
string url)
    {
        // Имя файла можно извлечь из URL, например
        string fileName = url.Substring(url.LastIndexOf("/") +
1);

        try
        {
            // Выполняем HTTP-запрос для загрузки файла
            byte[]          fileData          =          await
httpClient.GetByteArrayAsync(url);

            // Сохраняем файл на диск
            System.IO.File.WriteAllBytes(fileName, fileData);

            Console.WriteLine($"Файл {fileName} загружен.");
        }
        catch (Exception ex)
        {
            Console.WriteLine($"Ошибка   при   загрузке   файла
{fileName}: {ex.Message}");
        }
    }
}
```

Контрольные вопросы

1. Дайте определение многопоточности.
2. Какие преимущества дает использование многопоточности в приложениях?
3. Опишите основные проблемы, которые могут возникнуть при использовании многопоточности, и предложите способы их решения.
4. В каких случаях использование многопоточности может замедлить работу приложения?
5. Какие инструменты и библиотеки предоставляет .NET для работы с потоками?
6. Опишите различия между классами Thread и Task в .NET.
7. Как синхронизировать доступ к общим ресурсам из разных потоков?
8. Какие существуют стратегии планирования выполнения потоков?
9. Как обрабатывать исключения в многопоточных приложениях?
10. Опишите модель памяти в .NET. Как она влияет на работу многопоточных приложений?

Лабораторная работа № 28

Тема: **«Сериализация и десериализация объектов. Работа с JSON».**

Цель: изучить процессы сериализации и десериализации объектов в C#, а также научиться работать с форматом JSON, чтобы хранить и передавать данные между различными компонентами программы и между системами.

Краткие теоретические сведения

1. Сериализация и десериализация объектов

Формат JSON в настоящее время – один из наиболее часто используемых текстовых форматов данных, применяющийся как для хранения информации об объектах, так и для обмена этой информацией по Сети. Рассмотрим один из вариантов сериализации/десериализации объектов C# с использованием классов и объектов, расположенных в пространстве имен `System.Text.Json`.

В соответствии с определением, данным Microsoft, сериализация – это процесс преобразования объекта в поток байтов для сохранения или передачи в память, базу данных или файл. Сериализация используется для того, чтобы сохранить состояния объекта для последующего воссоздания при необходимости. Обратный процесс называется десериализацией.

Для сериализации/десериализации JSON в C# можно использовать как штатный сериализатор `JsonSerializer`, расположенный в пространстве имен `System.Text.Json`, так и решения сторонних разработчиков, например, часто используемый JSON.NET. Будем использовать штатный сериализатор.

2. Пример сериализации и десериализации JSON

Допустим, у нас имеется вот такой класс для хранения информации о человеке:

```
public class Person
{
public string Name { get; set; }
public string Surname { get; set; }
private int age;
public int Age
```

```
{
get => age;
set
{
if ((value < 0) || (value > 100))
throw new Exception("Возраст должен находится в интервале от 0
до 100 лет");
else
age = value;
}
}
public DateTime Birthday{ get; set; }
}
```

Создадим объект этого класса и сохраним состояние объекта в файл в формате JSON:

```
Person person = new Person()
{
Name = "Вася",
Surname = "Пупкин",
Age = 38,
Birthday = new DateTime(1983, 01, 16)
};
string personJson = JsonSerializer.Serialize(person,
typeof(Person));
StreamWriter file = File.CreateText("person.json");
file.WriteLine(personJson);
file.Close();
```

Здесь выполнили следующие действия:

- Создали сам объект для сериализации (person)

- Сериализовали объект в строку (string) используя сериализатор JsonSerializer

- Записали полученную строку в текстовый файл person.json.

Теперь рядом с exe-файлом приложения появился файл со следующим содержимым:

```
{"Name":"\u0412\u0430\u0441\u044F","Surname":"\u041F\u0443\u04
3F\u043A\u0438\u043D","Age":38,"Birthday":"1983-01-
16T00:00:00"}
```

Теперь можно при следующем запуске программы восстановить (десериализовать) этот объект в нашей программе, например, так:

```
if (File.Exists("person.json"))
{
```

```
string data = File.ReadAllText("person.json");
Person person = JsonSerializer.Deserialize<Person>(data);
Console.WriteLine($"Имя: {person.Name}");
Console.WriteLine($"Фамилия: {person.Surname}");
Console.WriteLine($"Возраст: {person.Age}");
Console.WriteLine($"День рождения: {person.Birthday}");
}
```

Результат:

```
Имя: Вася
Фамилия: Пупкин
Возраст: 38
День рождения: 16.01.1983 0:00:00
```

3. Параметры сериализации

Для настройки сериализации можно передать в метод `Serialize` второй параметр – объект класса `JsonSerializerOptions`, который содержит настройки сериализации/десериализации объектов. Основные свойства этого класса следующие:

`JavaScriptEncoder Encoder` – возвращает или устанавливает кодировщик, используемый при экранировании строк. Укажите значение null для использования кодировщика по умолчанию;

`bool IgnoreReadOnlyProperties` – аналогично устанавливает, будут ли сериализоваться свойства, предназначенные только для чтения;

`int MaxDepth` – возвращает или задает максимальную глубину, разрешенную при сериализации или десериализации JSON, при этом значение по умолчанию `0` указывает максимальную глубину 64;

`bool PropertyNameCaseInsensitive` – возвращает или задает значение, которое определяет, использует ли имя свойства сравнение без учета регистра во время десериализации. Значение по умолчанию – false;

`bool WriteIndented` – устанавливает, будут ли добавляться в `json` пробелы (условно говоря, для красоты). Если равно `true` устанавливаются дополнительные пробелы.

Рассмотрим несколько примеров использования этих настроек сериализации Json.

Как получить форматированную строку (с лидирующими пробелами) Json?

Для этого необходимо в настройках сериализации указать значение WriteIndented = true:

```
Person person = new Person()
{
Name = "Вася",
Surname = "Пупкин",
Age = 38,
Birthday = new DateTime(1983, 01, 16)
};
JsonSerializerOptions options = new JsonSerializerOptions()
{
WriteIndented = true
};
string personJson = JsonSerializer.Serialize(person, options);
Console.WriteLine(personJson);
```

Результат

```
{
   "Name": "\u0412\u0430\u0441\u044F",
   "Surname": "\u041F\u0443\u043F\u043A\u0438\u043D",
   "Age": 38,
   "Birthday": "1983-01-16T00:00:00"
}
```

Как запретить экранировать строки при сериализации JSON?

В примере выше у нас все строки (имя, фамилия) экранированы. С точки зрения безопасности – это полностью правильное решение, но не всегда удобно для чтения.

Чтобы позволить сериализатору не экранировать символы в строках, можно воспользоваться свойством Encoder у JsonSerializerOptions следующим образом:

```
using System.Text.Encodings.Web;
JsonSerializerOptions options = new JsonSerializerOptions()
{
WriteIndented = true, //добавляем пробелы для красоты
Encoder = JavaScriptEncoder.UnsafeRelaxedJsonEscaping //не
экранируем символы в строках
};
```

В этом случае, объект из примера выше будет выглядеть следующим образом:

```
{
"Name": "Вася",
"Surname": "Пупкин",
"Age": 38,
"Birthday": "1983-01-16T00:00:00"
}
```

4. Настройка сериализации с помощью атрибутов

При сериализации/десериализации объектов в JSON бывает необходимым исключить какое-либо свойство из строки JSON. Например, в нашем объекте Person таким свойством может быть свойство Age, так как можно легко рассчитать возраст человека, зная его день рождения и каждый раз «таскать» это свойство в JSON – смысла нет. В этом случае нам может пригодиться настройка сериализации JSON с использованием атрибутов.

Для того, чтобы воспользоваться таким видом настройки, необходимо подключить пространство имен `System.Text.Json.Serialization`.

Исключим свойство Age из сериализуемых, используя атрибут JsonIgnore:

```
public class Person
{
public string Name { get; set; }
public string Surname { get; set; }
private int age;
[JsonIgnore]
public int Age //теперь свойство Age не будет сериализоваться
{
get => age;
set
{
if ((value < 0) || (value > 100))
throw new Exception("Возраст должен находится в интервале от 0
до 100 лет");
else
age = value;
}}
public DateTime Birthday { get; set; }
}
```

Также, часто используемым атрибутом для настройки сериализации/десериализации JSON является атрибут JsonPropertyName, который позволяет задать произвольное имя сериализуемому свойству. Например,

```
public class Person
{
[JsonPropertyName("PersonFamily")]
public string Surname { get; set; }
}
```

Теперь свойство Surname будет сериализоваться в JSON с именем PersonFamily.

5. Особенности сериализации и десериализации JSON

При сериализации/десериализации JSON с использованием штатного сериализатора JsonSerializer необходимо обратить внимание на следующие моменты:

Десериализуемый объект должен иметь конструктор без параметров.

Например, во всех примерах по работе с JSON в C# использовали конструктор по умолчанию. Если бы наш класс имел, например, только вот такой конструктор:

```
public class Person
{
public string Name { get; set; }
....
public Person(string name)
{
Name = name;
}
}
```

То при попытке десериализовать такой объект получим исключение:

```
System.NotSupportedException: «Deserialization of reference types without parameterless constructor is not supported. Type ‘CSharpJson.Person'» (Сериализации подлежат только публичные свойства. Все непубличные свойства, а также поля сериализуемого класса игнорируются сериализатором).
```

6. Сериализация производных классов

Итак, вернемся к нашему классу Person, который выглядит вот так:

```
public class Person
{
public string Name { get; set; }
[JsonPropertyName("PersonFamily")]
public string Surname { get; set; }
public int age;
[JsonIgnore]
public int Age //теперь свойство Age не будет сериализоваться
{
get => age;
set
{
if ((value < 0) || (value > 100))
throw new Exception("Возраст должен находится в интервале от 0
до 100 лет");
else
age = value;
}
}
public DateTime Birthday { get; set; }
public Person(string name)
{
Name = name;
}
public Person()
{ }
}
```

С помощью атрибутов настроили сериализацию, а также объявили конструктор без параметров, чтобы объект мог спокойно десериализоваться из JSON.

Теперь попробуем создать производный класс, например, описывающий аккаунт пользователя:

```
public class Account : Person
{
public string Login { get; set; }
public string Password { get; set; }
}
```

Если попробуем сериализовать объект такого класса, то никаких проблем не будем наблюдать – настройки сериализации свойств сохранятся, и увидим ровно то, что и ожидаем увидеть:

```
Account account = new Account()
{
Name = "Вася",
Surname = "Пупкин",
Age = 38,
Birthday = new DateTime(1983, 01, 16),
Login = "Uasya",
Password = "qwerty"
};
JsonSerializerOptions options = new JsonSerializerOptions()
{
WriteIndented = true,
Encoder = JavaScriptEncoder.UnsafeRelaxedJsonEscaping
};
string personJson = JsonSerializer.Serialize(account, options);
Console.WriteLine(personJson);
```

Результат:

```
{
"Login": "Uasya",
"Password": "qwerty",
"Name": "Вася",
"PersonFamily": "Пупкин",
"Birthday": "1983-01-16T00:00:00"
}
```

`JsonSerializer` поместил свойства производного класса (`Account`) перед свойствами базового класса, но результат ожидаемый. Проблемы с сериализацией производных классов в JSON начинаются в том случае, если попробуем применить при работе с нашим новым классом полиморфизм.

Например, попробуем сериализовать вот такой объект:

```
Person account = new Account() //фактически, account - это
объект типа Account
{
Name = "Вася",
Surname = "Пупкин",
Age = 38,
Birthday = new DateTime(1983, 01, 16),
Login = "Uasya",
Password = "qwerty"
};
```

Результат сериализации будет такой:

```
{
"Name": "Вася",
"PersonFamily": "Пупкин",
"Birthday": "1983-01-16T00:00:00"
}
```

Сериализовались только свойства базового класса, несмотря на то что переменная `account` – это, фактически, объект типа `Account`. Такое поведение сериализатора `JsonSerializer` предназначено для предотвращения случайного доступа к данным в производном типе, созданном во время выполнения. Чтобы сериализовать объект производного класса можно воспользоваться нижеследующими подходами.

7. Полиморфная сериализация JSON

Чтобы сериализовать в JSON объект производного типа, можно воспользоваться одной из перегрузок метода `Serialize`, явно указав тип сериализуемого объекта:

```
string accountJson = JsonSerializer.Serialize(account,
typeof(Account), //явно указали какой тип содержится в account
options);
```

В этом случае, результат сериализации будет полным, т.е. в строку JSON попадут все свойства класса `Account`.

Указываем тип сериализуемого объекта как object

```
string account2Json = JsonSerializer.Serialize<object>(account,
options);
```

В этом случае, как и в предыдущем, строка JSON будет содержать все сериализуемые свойства производного класса:

```
{
"Login": "Uasya",
"Password": "qwerty",
"Name": "Вася",
"PersonFamily": "Пупкин",
"Birthday": "1983-01-16T00:00:00"
}
```

8. Как создать класс C# из JSON в онлайн

В Сети можно также встретить достаточно много различных онлайн-сервисов, позволяющих создавать из JSON классы C#, например, https://json2csharp.com (рис. 28.1).

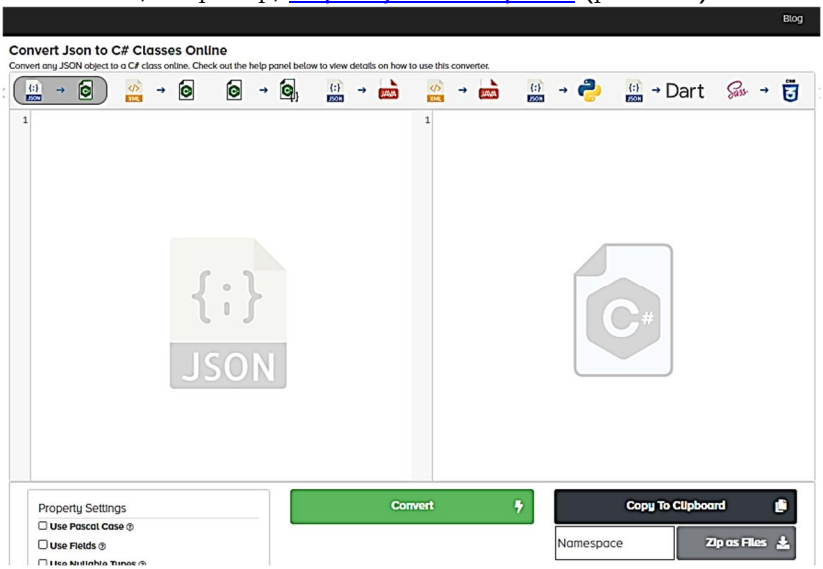

Рис. 28.1. Веб-интерфейс для создания класса из JSON

Основная задача данного сервиса – по полученному JSON сформировать один или несколько классов C#. При этом в json2csharp.com можно также задать настройки преобразования, например, добавить к свойствам классов атрибуты `JsonProperty` для настройки сериализации и десериализации.

Задания и порядок выполнения работы

Задание 1. Выполните все примеры описанные в теоретической части

Задание 2. Реализуйте класс `Person` с полями `name`, `age` и `hobbies`, а также методы для установки и получения значений этих полей. Сериализуйте и десериализуйте объект этого класса из/в JSON.

Задание 3. Реализуйте метод для форматирования JSON-строк с использованием методов `ToString` и

JsonConvert.SerializeObject. Сравните результаты и определите, какой из методов более предпочтителен для различных типов данных.

Задание 4. Разработайте Windows-приложение, которое позволяет сохранять и загружать данные в/из JSON-файла. Пользователь должен иметь возможность добавлять, изменять и удалять записи в JSON-файле.

Задание 5. Выполните сериализацию свойств Windows Forms приложений, которые были разработаны в лабораторной работе №20.

Контрольные вопросы

1. Дайте определение процессам сериализации и десериализации данных в контексте программирования?
2. Что такое формат JSON и для каких целей он используется?
3. Опишите алгоритм сериализации объекта в C# и приведите пример кода?
4. В чем разница между методами ToString() и JsonConvert.SerializeObject() при работе с JSON?
5. Какие особенности необходимо учитывать при сериализации объектов со ссылочными типами данных?
6. Опишите процесс десериализации JSON-строки в объект с помощью библиотеки Newtonsoft.Json и приведите соответствующий пример кода?
7. В чем отличие между сериализацией и форматированием объектов в JSON?
8. Какие преимущества предоставляет работа с JSON по сравнению с другими форматами данных при разработке .NET-приложений?
9. Какие типы данных могут быть преобразованы в формат JSON без потери информации?
10. Опишите, как можно проверить корректность JSON-данных, полученных из ненадежных источников.

Лабораторная работа № 29

Тема: **«Работа с датой и временем в C#»**.

Цель: изучить различные способы работы с датой и временем в C#, а также сравнить их производительность и удобство использования в различных сценариях.

Краткие теоретические сведения

1. Объект DateTime

Работа с датой и временем в C#, в основном, проводится с использованием структуры DateTime.

Для создания объекта типа DateTime можно воспользоваться одним из конструкторов структуры:

```
public DateTime()
public DateTime(int year, int month, int day)
public DateTime(int year, int month, int day, Calendar calendar)
public DateTime(int year, int month, int day, int hour, int
minute, int second)
public DateTime(int year, int month, int day, int hour, int
minute, int second, DateTimeKind kind)
public DateTime(int year, int month, int day, int hour, int
minute, int second, Calendar calendar)
public DateTime(int year, int month, int day, int hour, int
minute, int second, int millisecond)
public DateTime(int year, int month, int day, int hour, int
minute, int second, int millisecond, DateTimeKind kind)
public DateTime(int year, int month, int day, int hour, int
minute, int second, int millisecond, Calendar calendar)
public DateTime(int year, int month, int day, int hour, int
minute, int second, int millisecond, Calendar calendar,
DateTimeKind kind)
public DateTime(long ticks)
public DateTime(long ticks, DateTimeKind kind)
```

Например:

```
DateTime empty = new DateTime();
DateTime date = new DateTime(2021,09,11);
DateTime dateTime = new DateTime(2021, 09, 11, 10, 59, 59);
Console.WriteLine(empty);
Console.WriteLine(date);
Console.WriteLine(dateTime);
```

В результате получим следующий вывод консоли:

```
01.01.0001 0:00:00
11.09.2021 0:00:00
11.09.2021 10:59:59
```

В первом случае мы использовали конструктор по умолчанию и получили минимально возможную дату и время, во втором случае –-задали только дату, и в третьем – дату и время.

При этом, если необходимо получить текущие значения даты и времени в C#, то можно воспользоваться следующими свойствами DateTime:

```
DateTime now = DateTime.Now;//текущие дата и время
DateTime nowUtc = DateTime.UtcNow; //текущие дата и время в UTC
DateTime dateUtc = DateTime.Today; //текущая дата
Console.WriteLine($"Now: {now}");
Console.WriteLine($"nowUtc: {nowUtc}");
Console.WriteLine($"Today {dateUtc}");
Now: 09.11.2021 19:27:16
nowUtc: 09.11.2021 13:27:16
Today 09.11.2021 0:00:00
```

2. Работа с DateTime в C#

Структура DateTime содержит также ряд полезных свойств и методов, позволяющих манипулировать значениями даты и времени в своих приложениях.

Для того, чтобы получить из объекта типа DateTime только дату или только время, необходимо воспользоваться его свойствами Date или TimeOfDay:

```
DateTime now = DateTime.Now;//текущие дата и время
Console.WriteLine($"Дата и время: {now}");
Console.WriteLine($"Дата: {now.Date}");
Console.WriteLine($"Время: {now.TimeOfDay}");
Дата и время: 09.11.2021 19:33:08
Дата: 09.11.2021 0:00:00
Время: 19:33:08.3884058
```

Обратите внимание, что время выводится вплоть до миллисекунд.

Чтобы получить порядковый номер дня в году, необходимо использовать свойство DayOfYear:

```
DateTime now = new DateTime(2021, 09, 13);
int day = now.DayOfYear;
if (day == 256)
Console.WriteLine($"День: {day}. Сегодня День программиста!");
День: 256. Сегодня День программиста!
```

Также можно воспользоваться другими свойствами Day.., чтобы получить порядковые номера дня:

```
DateTime now = new DateTime(2021, 09, 13);
Console.WriteLine($"Порядковый номер дня в неделе:
{(int)now.DayOfWeek}");
Console.WriteLine($"Порядковый номер дня в месяце: {now.Day}");
Console.WriteLine($"Порядковый номер дня в году:
{now.DayOfYear}");
```

Свойство `DayOfWeek` представляет собой тип `enum` (перечисление), которое может принимать следующие значения:

```
public enum DayOfWeek
{
Sunday,
Monday,
Tuesday,
Wednesday,
Thursday,
Friday,
Saturday
}
```

Структура `DateTime` содержит ряд методов, позволяющих манипулировать значениями даты и времени:

```
public DateTime Add(TimeSpan value)
public DateTime AddDays(double value)
public DateTime AddHours(double value)
public DateTime AddMilliseconds(double value)
public DateTime AddMinutes(double value)
public DateTime AddMonths(int months)
public DateTime AddSeconds(double value)
public DateTime AddTicks(long value)
public DateTime AddYears(int value)
```

Обратите внимание, что большинство параметров в этих методах имеют тип данных `double`, что, технически, позволяет нам прибавлять к значениям даты и времени, например пол дня или пол часа:

```
DateTime now = new DateTime(2021, 09, 13);
Console.WriteLine($"Прибавим 1 день: {now.AddDays(1)}");
Console.WriteLine($"Прибавим пол часа: {now.AddHours(0.5)}");
Console.WriteLine($"Прибавим один год: {now.AddYears(1)}");
Прибавим 1 день: 14.09.2021 0:00:00
Прибавим пол часа: 13.09.2021 0:30:00
Прибавим один год: 13.09.2022 0:00:00
```

При необходимости, можно передавать в методы `Add...` и отрицательные значения – тип `double` это позволяет. В результате

будем не прибавлять, а отнимать определенное значение из DateTime

Для вычитания из одного объекта DateTime другого объекта DateTime можно воспользоваться методом Subtract:

```
DateTime date1 = new DateTime(2021, 09, 13);
DateTime date2 = new DateTime(2020, 09, 13); //дата на год меньше
Console.WriteLine($"date1.Subtract: {date1.Subtract(date2)}");
```

Результатом выполнения метода Subtract является объект типа TimeSpan. В результате получим:

```
date1.Subtract: 365.00:00:00
```

3. Форматирование дат в C#

Структура DateTime также содержит ряд методов, позволяющих форматировать значение DateTime и получать результат в виде строки:

```
DateTime date = DateTime.Now;
Console.WriteLine(date.ToLongDateString());
Console.WriteLine(date.ToLongTimeString());
Console.WriteLine(date.ToShortDateString());
Console.WriteLine(date.ToShortTimeString());
9 ноября 2021 г.
20:41:31
09.11.2021
20:41
```

4. Измерение времени выполнения операции в C#

При разработке различных программ иногда бывает необходимо измерить точное время какой-либо операции, например, узнать сколько времени требуется на загрузку данных из файла, запись в базу данных и так далее. Рассмотрим то, как измерить время выполнения операции в C#, используя стандартные средства и возможности языка.

Пространство имен System.Diagnostics содержит классы, позволяющие нашим приложениям взаимодействовать с журналами событий, системными процессами и счётчиками производительности. В числе прочего, это пространство имен содержит класс под названием Stopwatch, который можно в своих

приложениях C# использовать для точного измерения затраченного времени.

Для начала рассмотрим простой пример использования класса `Stopwatch` для измерения затраченного времени на выполнение операции.

```
namespace StopwatchExample
{
class Program
{
static void Main(string[] args)
{
//создаем объект
Stopwatch stopwatch = new Stopwatch();
//засекаем время начала операции
stopwatch.Start();
//выполняем какую-либо операцию
for (int i = 0; i < 10001; i++)
{
Console.WriteLine(i);
}
//останавливаем счётчик
stopwatch.Stop();
//смотрим сколько миллисекунд было затрачено на выполнение
Console.WriteLine(stopwatch.ElapsedMilliseconds);
}
}
}
```

Чтобы измерить время выполнения операции в C# нам необходимо выполнить несколько простых шагов:

1. Создать объект класса `Stopwatch`;

2. Выполнить метод `Start()` для того, чтобы засечь время начала операции;

3. Выполнить метод `Stop()` для того, чтобы засечь время окончания операции;

4. Воспользоваться одним из свойств объекта для получения данных о затраченном на выполнение операции времени.

В примере использовано свойство `ElapsedMilliseconds`, которое позволяет получить количество миллисекунд, затраченных на выполнение операции.

5. Свойства Stopwatch

Свойство `Elapsed` позволяет получить общее затраченное время, измеренное текущим экземпляром класса `Stopwatch`. Описание свойства выглядит следующим образом:

```
public TimeSpan Elapsed { get; }
```

Свойство возвращает объект типа `TimeSpan` – интервал времени, используя который можно получить время выполнения операции в удобном для вас виде. Например,

```
//получаем объект TimeSpan
TimeSpan ts = stopwatch.Elapsed;
// Создаем строку, содержащую время выполнения операции.
string elapsedTime =
String.Format("{0:00}:{1:00}:{2:00}.{3:00}",
ts.Hours, ts.Minutes, ts.Seconds,
ts.Milliseconds / 10);
Console.WriteLine(elapsedTime);
ElapsedMilliseconds
```

Свойство `ElapsedMilliseconds` позволяет получить общее затраченное время, измеренное текущим экземпляром класса `Stopwatch` в миллисекундах. В примере использования класса `Stopwatch` выше продемонстрировано использование этого свойства.

Свойство `ElapsedTicks` позволяет получить общее время выполнения операции в тактах таймера, измеренное текущим экземпляром `Stopwatch`. Такт – это наименьшая единица времени, которую `Stopwatch` может измерять таймер. В следующем примере показано использование свойства `ElapsedTicks` для измерения времени, затраченного на преобразование строки в целое число типа `int`.

```
int num;
//создаем объект
Stopwatch stopwatch = new Stopwatch();
//засекаем время начала операции
stopwatch.Start();
num = int.Parse("135");
//останавливаем счётчик
stopwatch.Stop();
Console.WriteLine($"num = {num}");
//смотрим сколько тактов было затрачено на выполнение
Console.WriteLine(stopwatch.ElapsedTicks);
```

Результатом выполнения этого кода может быть вот такой вывод консоли:

```
num = 135
16878
IsRunning
```

Свойство `IsRunning` позволяет получить значение типа `bool`, указывающее на то запущен ли в данный момент таймер `Stopwatch`.

6. Поля Stopwatch

Класс `Stopwatch` содержит два статических поля, позволяющих получить сведения о настройках таймера.

Поле `Frequency` содержит частоту таймера в виде количества тактов в секунду.

```
public static readonly long Frequency;
```

Это поле удобно использовать вместе со свойством `ElapsedTicks` для преобразования количества тактов в секунды. Например,

```
int num;
long freq = Stopwatch.Frequency; //частота таймера
Stopwatch stopwatch = new Stopwatch();
stopwatch.Start();
num = int.Parse("135");
//останавливаем счётчик
stopwatch.Stop();
double sec = (double)stopwatch.ElapsedTicks / freq; //переводим
такты в секунды
Console.WriteLine($"num = {num} \r\n Частота таймера {freq}
такт/с \r\n Время в тактах {stopwatch.ElapsedTicks} \r\n Время
в секундах {sec}");
```

Учитывая особенности деления целых чисел в C# для того, чтобы получить конкретное значение секунд нам потребовалось привести одно из значений (в данном случае значение свойства `ElapsedTicks`) к типу `double`.

Свойство `IsHighResolution` указывает, зависит ли таймер `Stopwatch` от счетчика производительности высокого разрешения (`true`) или же использует класс `DateTime` (`false`).

```
public static readonly bool IsHighResolution;
```

Пример использования поля

```
class Program
```

```
{
static void Main(string[] args)
{
DisplayTimerProperties();
}
public static void DisplayTimerProperties()
{
if (Stopwatch.IsHighResolution)
{
Console.WriteLine("Операции рассчитываются с использованием
системного счетчика производительности с высоким
разрешением.");
}
else
{
Console.WriteLine("Операции рассчитываются с использованием
класса DateTime.");
}
long frequency = Stopwatch.Frequency;
Console.WriteLine($" Частота таймера = {frequency}");
long nanosecPerTick = (1000L * 1000L * 1000L) / frequency;
Console.WriteLine($" Таймер работает с точностью до
{nanosecPerTick} наносекунд");
}
}
```

Вывод консоли будет иметь следующий вид:

```
Операции рассчитываются с использованием системного счетчика
производительности с высоким разрешением.
Частота таймера = 10000000
Таймер работает с точностью до 100 наносекунд
```

7. Методы Stopwatch

Рассмотрим основные методы класса `Stopwatch`, которые можно использовать для измерения точного времени выполнения операции в C#.

Метод `Start()` запускает или возобновляет работу таймера `Stopwatch`. В свою очередь, `Stop()` выполняет противоположную операцию – останавливает работу таймера. Использование этих методов продемонстрировано в самом первом примере из этой статьи.

Метод `StartNew()` выполняет сразу несколько операций – он инициализирует новый экземпляр класса `Stopwatch`, обнуляет счётчик затраченного времени и запускает таймер. То есть, этот

метод позволяет немного сократить исходный код программы. Например, код из первого примера можно было бы записать вот так:

```
Stopwatch stopwatch = Stopwatch.StartNew();//создаем и
запускаем таймер
for (int i = 0; i < 10001; i++)
{
Console.WriteLine(i);
}
//останавливаем счётчик
stopwatch.Stop(); //смотрим сколько миллисекунд было затрачено
на выполнение
Console.WriteLine(stopwatch.ElapsedMilliseconds);
```

Метод `Reset()` останавливает измерение интервала времени и обнуляет счётчик затраченного времени. Использование `Reset()` позволяет избежать создания новых экземпляров `Stopwatch` для измерения времени, затраченного на выполнение нескольких операций в C#.

```
Stopwatch stopwatch = Stopwatch.StartNew();//создаем и
запускаем таймер
for (int i = 0; i < 100; i++)
{
Console.WriteLine(i);
}
//останавливаем счётчик
stopwatch.Stop(); //смотрим сколько миллисекунд было затрачено
на выполнение
Console.WriteLine($"Первая операция
{stopwatch.ElapsedMilliseconds}");
stopwatch.Reset(); //сбросили счётчик
stopwatch.Start(); //запустили счётчик
for (int i = 0; i < 100; i++)
{
Console.WriteLine(i*i);
}
stopwatch.Stop(); //смотрим сколько миллисекунд было затрачено
на выполнение
Console.WriteLine($"Вторая операция
{stopwatch.ElapsedMilliseconds}");
```

Метод `Restart()` останавливает измерение интервала времени, обнуляет затраченное время и повторно запускает таймер. Таким образом, предыдущий пример можно переписать следующим образом:

```
Stopwatch stopwatch = Stopwatch.StartNew();//создаем и
запускаем таймер
for (int i = 0; i < 100; i++)
{
Console.WriteLine(i);
}
//останавливаем счётчик
stopwatch.Stop(); //смотрим сколько миллисекунд было затрачено
на выполнение
Console.WriteLine($"Первая операция
{stopwatch.ElapsedMilliseconds}");
stopwatch.Restart(); //перезапускаем счётчик
for (int i = 0; i < 100; i++)
{
Console.WriteLine(i*i);
}
stopwatch.Stop(); //смотрим сколько миллисекунд было затрачено
на выполнение
Console.WriteLine($"Вторая операция
{stopwatch.ElapsedMilliseconds}");
```

Таким образом, класс `Stopwatch` из пространства имен `System.Diagnostics` C# позволяет измерить время выполнения операции с точностью до 100 наносекунд в зависимости от того, что используется для работы с интервалами времени – таймер высокого разрешения или же класс `DateTime`.

Задания и порядок выполнения работы

Задание 1. Повторите все примеры из раздела кратких теоретических сведений.

Задание 2. Напишите программу, в которой пользователь вводит дату своего рождения, а программа вычисляет, сколько прошло полных лет, месяцев и дней от указанной даты до текущей.

Задание 3. Напишите программу, в которой для указанного интервала времени (в годах) определяются годы, первый день которых (1 января) попадает на понедельник.

Задание 4. Напишите программу, содержащую статический метод, которому передаются два аргумента, определяющих некоторые даты. Метод сравнивает эти даты на предмет совпадения. Даты считаются совпадающими, если они отличаются не более чем на определенный интервал времени. Интервал времени задается третьим аргументом метода.

Контрольные вопросы

1. Какие основные функции и методы используются для работы с датой и временем в C#?

2. Как создать объект даты и времени в C# и какие параметры нужно задать?

3. Опишите различные способы форматирования даты и времени в C#.

4. Как сравнивать даты и времена в C# с учетом часового пояса и других факторов?

5. Что такое структуры `DateTime` и `DateTimeOffset` и в чем их различие?

6. Как использовать классы `TimeSpan` и `TimeZoneInfo` для работы с временными интервалами и часовыми поясами в C#?

7. Как управлять локалью (языком и форматом даты и времени) при работе с датой и временем в C#?

8. Как обрабатывать даты, выходящие за пределы диапазона, который может быть представлен с помощью структур даты и времени в .NET?

9. Опишите использование делегатов и событий для отслеживания изменений даты и времени.

10. Как работать с датой и временем, хранящимися в разных форматах (строки, базы данных, JSON и т.д.) в C#?

Лабораторная работа № 30

Тема: **«Параллельное программирование»**.

Цель: освоить основы параллельного программирования на языке C# с использованием библиотеки TPL (Task Parallel Library) для повышения производительности и эффективного использования многоядерных процессоров.

Краткие теоретические сведения

Основная цель Task Parallel Library (TPL, библиотека параллельных задач) – повышение производительности разработчиков при добавлении многопоточности в приложения C#. Начиная с .NET Framework 4 наиболее предпочтительным вариантом использования многопоточности в приложении является использование именно библиотек TPL, но, при этом, такие классы как Thread, также могут использоваться и все также находят широкое применение на сегодняшний день.

1. Параллельное выполнение задач

До появления в составе .NET библиотеки TPL, чтобы организовать параллельное выполнение нескольких задач разработчику необходимо было как минимум:

1. Организовать пул потоков для их управления.

2. Использовать различные средства для синхронизации потоков.

Библиотека TPL позволяет, в основном, освободить разработчика от рутинных операций при работе с потоками, в частности, TPL позволяет:

1. Динамически масштабировать степень параллелизма для наиболее эффективного использования всех доступных процессоров

2. Осуществить секционирование работы,

3. Осуществить планирование потоков в пуле ThreadPool,

4. Осуществить поддержку отмены, управления состоянием и других низкоуровневых задачи.

2. Задача (Task)

Основная концепция библиотеки TPL – использование задач (класс Task располагается в пространстве имен System.Threading.Tasks).

Задача – это какая-либо отдельная операция, которая должна выполняться параллельно.

В зависимости от потребностей, можно запускать задачи как в отдельном потоке из пула потоков, так и в главном потоке приложения (синхронно). Рассмотрим различные варианты запуска задач в C#.

2.1. Использование метода Start

Метод Start запускает выполнение предварительно созданной задачи:

```
using System;
using System.Threading.Tasks;
namespace Tasks
{
internal class Program
{
static void Main(string[] args)
{
Task task = new Task(Counter);
task.Start();
Console.ReadLine();
}
public static void Counter()
{
for (int i = 0; i < 10; i++)
Console.WriteLine(i);
Console.WriteLine("Счётчик закончил свою работу");
}}}
```

Для создания новой задачи (Task) используем конструктор, который в качестве параметра принимает делегат Action:

```
public delegate void Action();
```

Т.е. в конструкторе можно предать любой метод, соответствующий сигнатуре делегата. Таким у нас является метод Counter. После запуска задачи на выполнение методом Start задача будет выполнена в фоновом потоке, который будет взят из

пула потоков. Убедиться в этом можно, используя свойства класса Thread, например, дописав метод Counter следующим образом:

```
public static void Counter()
{
Console.WriteLine($"ID потока:
{Thread.CurrentThread.ManagedThreadId}");
Console.WriteLine($"Фоновый поток:
{Thread.CurrentThread.IsBackground}");
Console.WriteLine($"Поток взят из пула потоков:
{Thread.CurrentThread.IsThreadPoolThread}");
for (int i = 0; i < 10; i++)
Console.WriteLine(i);
}
```

Теперь запустим приложение и посмотрим на значения свойств текущего потока:

```
ID потока: 4
Фоновый поток: True
Поток взят из пула потоков: True
```

2.2. Использование статического метода Task.Factory.StartNew

В этом случае также передаем в метод StartNew делегат типа Action:

```
Task task = Task.Factory.StartNew(Counter);
```

Результат будет тот же, что и в предыдущем примере – задача запустится в отдельном фоновом потоке.

2.3. Использование статического метода Run

```
Task task = Task.Run(Counter);
```

Все три способа запуска задач, по сути, идентичны – выполнение задачи осуществляется в отдельном фоновом потоке из пула потоков. Четвертый способ отличается тем, что задача запускается синхронно в главном потоке приложения.

2.4. Использование метода RunSynchronously

Метод RunSynchronously выполняет запуск задачи в главном потоке приложения синхронно:

```
using System;
using System.Threading.Tasks;
using System.Threading;
namespace Tasks
```

```
{
internal class Program
{
static void Main(string[] args)
{
Task task = new Task(Counter);
task.RunSynchronously(); //запуск задачи синхронно
Console.ReadLine();
}
public static void Counter()
{
Console.WriteLine($"ID потока:
{Thread.CurrentThread.ManagedThreadId}");
Console.WriteLine($"Фоновый поток:
{Thread.CurrentThread.IsBackground}");
Console.WriteLine($"Поток взят из пула потоков:
{Thread.CurrentThread.IsThreadPoolThread}");
for (int i = 0; i < 10; i++)
Console.WriteLine(i);
}}}
```

В этом случае в консоли увидим следующие значения свойств потока, в котором запущена задача:

```
ID потока: 1
Фоновый поток: False
Поток взят из пула потоков: False
```

3. Ожидание задач

Рассмотрим следующий пример, который запускает задачу, используя метод Start:

```
static void Main(string[] args)
{
Task task = new Task(()=>Console.WriteLine("Привет Task"));
task.Start();
Console.WriteLine("Приложение завершило свою работу");
}
```

В результате можно наблюдать вот такой вывод консоли:

```
Приложение завершило свою работу
Привет Task
```

Строка «Приложение завершило свою работу» говорит нам о том, что главный поток приложения уже дошел до конца метода Main, но, при этом, сама задача была выполнена позже, хотя Start мы вызывали до вывода в консоль строки из главного потока. Такое поведение связано, в первую очередь, с самой организацией

многопоточной работы в .NET – операционная система сама решает, в зависимости от наличия ресурсов, какой поток будет работать в данный момент времени. Для большей наглядности, приведем ещё один пример:

```
using System;
using System.Threading.Tasks;
using System.Threading;
namespace Tasks
{
internal class Program
{
static void Main(string[] args)
{
Task task = new Task(Counter);
task.Start();
}
public static void Counter()
{
for (int i = 0; i < 10; i++)
Console.WriteLine(i);
}}}
```

Ожидаем, что на экран будут выведены числа с 0 до 9, а по факту – приложение завершит свою работу, не выведя на экран ни одного числа. Такое поведение, связано оно с тем, что поток, в котором выполняется задача, является фоновым (IsBackground=True). Чтобы задача из примера выше была выполнена, нам необходимо выполнить ожидание задачи. В библиотеке TPL это делается следующим образом:

```
using System;
using System.Threading.Tasks;
using System.Threading;
namespace Tasks
{
internal class Program
{
static void Main(string[] args)
{
Task task = new Task(Counter);
task.Start();
task.Wait();//ожидаем выполнение задачи
}
public static void Counter()
{
for (int i = 0; i < 10; i++)
```

```
Console.WriteLine(i);
}}}
```

Теперь можно запустить приложение и убедиться, что на экран будут выведены цифры от 0 до 9 и только после этого приложение завершит свою работу.

4. Работа с Task. Ожидание задач. Свойства и методы класса Task

4.1. Ожидание всего списка задач (метод WaitAll)

Когда необходимо, чтобы какая-либо задача запускалась только после того, как все задачи из списка (или массива) будут выполнены, удобно воспользоваться методом WaitAll:

```
using System;
using System.Threading.Tasks;
using System.Threading;
using System.Collections.Generic;
namespace Tasks
{
internal class Program
{
static void Main(string[] args)
{
List<Task> tasks = new List<Task>();
for (int i = 0; i < 10; i++)
{
tasks.Add(Task.Run(() =>
{
for (int i = 0; i < 100; i++)
Console.WriteLine($"Задача
{Thread.CurrentThread.ManagedThreadId} итерация {i}");
}));
}
Task.WaitAll(tasks.ToArray());//ожидаем пока все задачи не
закончат свою работу
Task task = new Task(() => { Console.WriteLine("Выполняем
завершающую задачу"); });
task.Start();
task.Wait();//ожидаем выполнение задачи
}}}
```

В этом примере создаем список (List) из десяти задач, каждая из которых выводит в консоль числа о 0 до 99. Задача task, которая создается самой последней и не входит в список будет выполнена только после того, как все задачи из списка tasks

будут выполнены. Для этого мы вызвали статический метод Task.WaitAll, в который передали массив созданных задач.

4.2. Ожидание любой задачи из списка (метод WaitAny)

Если нам нет необходимости ожидать все задачи из списка, а достаточно дождаться завершения одной любой задачи, то можно воспользоваться методом WaitAny. Перепишем наш пример таким образом, чтобы задача task запускалась сразу после того, как будет завершена хотя бы одна задача из списка tasks:

```
using System;
using System.Threading.Tasks;
using System.Threading;
using System.Collections.Generic;
namespace Tasks
{
internal class Program
{
static void Main(string[] args)
{
List<Task> tasks = new List<Task>();
for (int i = 0; i < 10; i++)
{
tasks.Add(Task.Run(() =>
{
for (int i = 0; i < 100; i++)
Console.WriteLine($"Задача
{Thread.CurrentThread.ManagedThreadId} итерация {i}");
}));
}
int finished = Task.WaitAny(tasks.ToArray());//ожидаем пока все
задачи не закончат свою работу
Console.WriteLine($"Первая задача, которая завершила свою
работу, находится в позиции {finished} в списке задач. ID
{tasks[finished].Id}");
Task task = Task.Run(() => { Console.WriteLine("Выполняем
завершающую задачу"); });
task.Wait();//ожидаем выполнение задачи
}}}
```

Метод WaitAny возвращает индекс первой завершенной задачи из переданного массива задач. Как только какая-либо задача из списка завершит свою работу, в консоль будет выведена запись, содержащая ID этой задачи и её индекс в списке. После этого будет запущена задача task.

5. Возвращение результатов выполнения задач

Кроме того, что задачи могут выполняться как процедуры (без возвращаемого результата), задача также может вернуть какой-либо результат. Для этого необходимо использовать универсальный класс `Task<TResult>`. Например, возьмем наш алгоритм определения простых чисел и реализуем его в виде задачи (`Task`). Пусть задача будет возвращать количество простых чисел:

```
using System;
using System.Threading.Tasks;
namespace Tasks
{
internal class Program
{
static void Main(string[] args)
{
Task<int> primeTask = new Task<int>(()=>TaskMethod(5000));
primeTask.Start();
Console.WriteLine($"В диапазоне от 1 до 5000 обнаружено
{primeTask.Result} простых чисел");
}
static int TaskMethod(int N)
{
int count = 0;
for (int i = 1; i <= N; i++)
{
if (IsPrime(i))
{
Console.Write($"{i} ");
count++;
}}
Console.WriteLine();
return count;
}
public static bool IsPrime(int number)
{
for (int i = 2; i < number; i++)
{
if (number % i == 0)
return false;
}
return true;
}}}
```

Задача primeTask возвращает значение типа int – количество найденных простых чисел в заданном диапазоне. Чтобы получить возвращаемое задачей значение используем её свойство Result. Аналогичным образом задача может возвращать любые необходимые нам объекты. Стоит обратить внимание на три момента:

1. Так как мы объявили задачу как Task<int>, то задача должна возвращать какое-либо целочисленное значение.

2. Метод, который определяем для задачи (TaskMethod) также должен возвращать объект типа int.

3. Как только обращаемся в свойство Result, приложение приостанавливает выполнение главного потока и возобновляет его работу только после того, как результат (Result) будет получен. Именно поэтому нам не потребовалось ожидать выполнение задачи и вызывать метод Wait.

6. Вложенные (дочерние) задачи

Дочерняя задача (или вложенная задача) – это экземпляр Task, который создается в делегате другой задачи, которая называется родительской задачей. Например, создадим дочернюю задачу:

```
using System;
using System.Threading.Tasks;
using System.Threading;
namespace Tasks
{
internal class Program
{
static void Main(string[] args)
{
Task mainTask = Task.Run(() =>
{
//создаем дочернюю задачу
Task.Run(() =>
{
Console.WriteLine("Выполняем дочернюю задачу");
Thread.Sleep(1000);
Console.WriteLine("Дочерняя задача выполнена");
});
Console.WriteLine("выполняем родительскую задачу");
}
```

```
);
mainTask.Wait();//ожидаем выполнения родительской задачи
Console.WriteLine("Задачи выполнены");
}}}
```

Так как порядок выполнения задач в примере не детерминирован, то в консоли можно увидеть такой результат:

```
Выполняем дочернюю задачу
выполняем родительскую задачу
Задачи выполнены
или такой
выполняем родительскую задачу
Выполняем дочернюю задачу
Задачи выполнены
```

Чтобы однозначно определить порядок выполнения родительских и дочерних задач, можно воспользоваться перегруженной версией конструктора для `Task` и присоединить дочернюю задачу к родительской. Перепишем пример следующим образом:

```
static void Main(string[] args)
{
Task mainTask = Task.Factory.StartNew(() =>
{
Console.WriteLine("Родительская задача начинает свою работу");
//создаем дочернюю задачу
var inner = Task.Factory.StartNew(() =>
{
Console.WriteLine("Выполняем дочернюю задачу");
Thread.Sleep(2000);
Console.WriteLine("Дочерняя задача выполнена");
}, TaskCreationOptions.AttachedToParent);//присоединяем задачу
}
);
mainTask.Wait();//ожидаем выполнения родительской задачи
Console.WriteLine("Задачи выполнены");
}
```

Здесь передали в конструкторе дочерней задачи второй параметр – `TaskCreationOptions.AttachedToParent`, который указывает, что дочерняя задача будет прикреплена к родительской. В этом случае родительская задача не завершит свою работу до тех пор, пока не будет выполнена дочерняя задача.

7. Задачи продолжения (continuation task)

Задачи продолжения (continuation task) или просто «продолжение» – это такие задачи, которые могут вызываться другой задачей (предшествующей) при завершении этой предшествующей задачи. Использование продолжений позволяет организовать цепочки взаимосвязанных задач и передавать некоторые результаты из одной задачи в другую.

7.1. Создание продолжения для одной задачи

В примере, представленном ниже, создается задача продолжения, которая выводит в консоль число, увеличенное на 1.

```
using System;
using System.Threading.Tasks;
namespace Tasks
{
internal class Program
{
static void Main(string[] args)
{
Task<int> task = new Task<int>(() => Increment(11) );//создаем
предшествующую задачу
Task                        continuation                        =
task.ContinueWith(x=>Display(task.Result));//создаем
продолжение
task.Start();//запускаем предшествующую задачу
continuation.Wait();//ожидаем окончание продолжения
}
static void Display(int x)
{
Console.WriteLine(x);
}
static int Increment(int a)
{
return ++a;
}}}
```

Вначале создаем предшествующую задачу task, которая должна возвращать нам результат типа int. Задача продолжения (continuation) принимает результат от task и выводит в консоль результат. Задача продолжения создается путем вызова метода ContinueWith предшествующей задачи. В качестве параметра, ContinueWith принимает делегат Action<Task<TResult>>.

В приведенном выше примере представлена простейшая цепочка, состоящая из двух задач: первая задача выполняет сложение, вторая – выводит результат в консоль.

7.2. Создание продолжения для нескольких задач

Если необходимо создать продолжение для нескольких задач, то можно воспользоваться одним из следующих статичных методов класса Task:

WhenAll – продолжение будет выполнено после завершения всех предшествующих задач;

WhenAny – продолжение будет выполнено после завершение какой-либо задачи из списка;

или аналогичными методами класса TaskFactory: ContinueWhenAll или ContinueWhenAny

Рассмотрим следующий пример:

```
using System;
using System.Collections.Generic;
using System.Threading.Tasks;
namespace Tasks
{
internal class Program
{
public static void Main()
{
var tasks = new List<Task<int>>();
for (int ctr = 1; ctr <= 10; ctr++)
{
int baseValue = ctr;
tasks.Add(Task.Factory.StartNew(b   =>   (int)b   *   (int)b,
baseValue));
}
Task<int[]> results = Task.WhenAll(tasks);
int sum = 0;
for (int ctr = 0; ctr <= results.Result.Length - 1; ctr++)
{
var result = results.Result[ctr];
Console.Write($"{result} {((ctr == results.Result.Length - 1) ?
"=" : "+")} ");
sum += result;
}
Console.WriteLine(sum);
}}}
```

Для создания задачи продолжения (results) вызывается статичный метод Task.WhenAll(). Продолжение отражает результаты десяти своих предшествующих задач, каждой из которых соответствует значение индекса в диапазоне от 1 до 10. Если предшествующие задачи завершаются успешно (их свойство Task.Status имеет значение RanToCompletion), то свойство Result продолжения представляет собой массив значений Task<TResult>.Result, возвращенных каждой предшествующей задачей. Затем рассчитывается сумма квадратов чисел от 1 до 10 и выводится в консоль:

```
1 + 4 + 9 + 16 + 25 + 36 + 49 + 64 + 81 + 100 = 385
```

8. Отмена задач

Отмена задач может потребоваться по различным причинам – долгое выполнение задач, отмена выполнения каких-либо действий в случае, если задача завершилась с ошибкой и так далее. В TPL предусмотрена возможность отмены параллельных задач с использованием специального объекта – CancellationToken.

8.1. Пример отмены параллельной задачи

В рассматриваемом ниже примере в параллельной задаче определяются простые числа. В случае, если пользователь вводит в консоль символ «N», выполнение задачи прерывается.

```
using System;
using System.Threading;
using System.Threading.Tasks;
namespace TaskCancel
{
internal class Program
{
static CancellationTokenSource cancelTokenSource = new
CancellationTokenSource();
static CancellationToken token = cancelTokenSource.Token;
static void Main(string[] args)
{
Task<int> primeTask = new Task<int>(() => TaskMethod(), token);
primeTask.Start();
Console.WriteLine("Введите N для отмены операции");
string s = Console.ReadLine();
```

```
if (s.ToUpper() == "N")
cancelTokenSource.Cancel();
Console.WriteLine($"Обнаружено {primeTask.Result} простых
чисел");
Console.Read();
}
static int TaskMethod()
{
int count = 0;
int i = 2;
while (!token.IsCancellationRequested)
{
if (IsPrime(i))
{
count++;
Console.Write($"{i} ");
}
i++;
Thread.Sleep(100);
}
Console.WriteLine($"Операция отменена.");
return count;
}
public static bool IsPrime(int number)
{
for (int i = 2; i < number; i++)
{
if (number % i == 0)
return false;
}
return true;
}}}
```

Для того, чтобы получить возможность прерывать выполнение параллельной задачи, нам необходимо создать токен отмены. Для этого мы вначале создаем объект:

```
static CancellationTokenSource cancelTokenSource = new
CancellationTokenSource();
```

И, затем, получаем сам токен:

```
static CancellationToken token = cancelTokenSource.Token;
```

Полученный токен передаем в конструктор задачи:

```
Task<int> primeTask = new Task<int>(() => TaskMethod(), token);
```

Чтобы задача была отменена, при выполнении какого-либо условия вызываем метод `Cancel()` объекта `CancellationTokenSource`. После того, как вызван метод `Cancel()`,

свойство токена token.IsCancellationRequested возвращает true, и в методе задачи происходит прерывание цикла while.

Как только пользователь вводит с клавиатуры символ «n» и жмет Enter, задача прерывается и в консоль выводится количество найденных простых чисел.

8.2. Прерывание «спящей» задачи

В рассмотренном выше примере вызывается уже известный метод Thread.Sleep(), который заставляет текущий поток уснуть на n миллисекунд. Если в примере написать следующее:

```
Thread.Sleep(10000);
```

Т.е. если усыпить поток на 10 секунд и попробовать остановить задачу с помощью токена отмены, то произойдет следующее: поток дойдет до строки с Thread.Sleep(10000), уснет на 10 секунд и только после этого задача будет отменена. Чтобы получить возможность остановить задачу со «спящим» потоком, можно использовать следующий подход:

```
static int TaskMethod()
{
int count = 0;
int i = 2;
while (!token.IsCancellationRequested)
{
//проверяем числа
var canceled = token.WaitHandle.WaitOne(10000);
if (canceled)
break;
// Thread.Sleep(10000);
}
Console.WriteLine($"Операция отменена.");
return count;
}
```

Свойство WaitHandle возвращает дескриптор WaitHandle, получающий сигнал при отмене токена. Класс WaitHandle инкапсулирует собственный обработчик синхронизации операционной системы и используется для представления всех объектов синхронизации в среде выполнения, которые допускают несколько операций ожидания. Метод WaitOne() блокирует текущий поток до получения сигнала объектом WaitHandle.

Теперь, если запустить пример и попробовать остановить задачу, то задача отменится сразу после того, как пользователь наберет «N» и нажмет Enter.

Таким образом, для отмены параллельных задач используется специальный объект класса CancellationToken, который необходимо передавать в конструкторе задач или в качестве параметра внешнего метода задачи. Кроме того, что объект типа CancellationToken может использоваться как обычный флаг (свойство IsCancellationRequested) отмены задачи, в этом объекте также имеется свойство WaitHandle, используя которое можно отменять задачи, в которых поток может останавливаться на определенное время.

9. Класс Parallel

Класс Parallel входит в состав библиотеки TPL .NET и позволяет достаточно легко (даже для неопытного разработчика) распараллелить ряд задач. В этом классе содержится ряд методов, обеспечивающих, в том числе, параллельное выполнение действий (Action) и параллельное выполнение итераций циклов for и foreach.

9.1. Метод Parallel.Invoke

Статический метод класса Parallel.Invoke() позволяет выполнить параллельно массив действий (Action). Например, запустим параллельное выполнение трех действий: расчёт факториала, определение простых чисел и вывод строки в консоль:

```
class Program
{
static void Factorial(int x)
{
Console.WriteLine($"Запускаем расчёт факториала числа {x}.
CurrentId = {Task.CurrentId}");
int result = 1;
for (int i = 1; i <= x; i++)
result *= i;
Console.WriteLine($"Расчёт факториала выполнен. Результат
{result}");
}
```

```
static void Prime(int x)
{
Console.WriteLine($"Запускаем расчёт простых чисел от 2 до {x}.
CurrentId = {Task.CurrentId}");
int total = 0;
for (int i = 2; i <= x; i++)
{
bool isPrime = true;
for (int j = 2; j < i; j++)
{
if (i % j == 0)
{
isPrime = false;
break;
}}
if (isPrime)
total++;
}
Console.WriteLine($"Расчёт простых чисел выполнен. Найдено
{total} чисел");
}
static void Main(string[] args)
{
Parallel.Invoke(() => Factorial(10), //первая задача
() => Prime(5000), //вторая задача
() => Console.WriteLine("Просто выводим строку на экран")
//третья задача
);
Console.ReadLine();
}}
```

Вывод консоли может быть следующим:

```
Запускаем расчёт факториала числа 10. CurrentId = 3
Расчёт факториала выполнен. Результат 3628800
Запускаем расчёт простых чисел от 2 до 5000. CurrentId = 1
Просто выводим строку на экран
Расчёт простых чисел выполнен. Найдено 669 чисел
```

Как мы уже знаем, пока не определим самостоятельно порядок выполнения параллельных вычислений, TPL нам не гарантирует, что задачи будут выполняться в том порядке, в котором они были определены. Вместе с этим, при использовании метода Parallel.Invoke() TPL сама возьмет на себя такие моменты как распределение задач по ядрам процессора, определение максимального числа потоков и так далее. У этого метода так же есть одна перегруженная версия, позволяющая производить

настройку выполнения задач, а также определить максимальное количество одновременно выполняемых задач:

```
public static void Invoke(ParallelOptions parallelOptions,
params Action[] actions);
```

9.2. Метод Parallel.For

Статический метод `Parallel.For` позволяет выполнять итерации цикла `for` параллельно. В самом простом варианте, сигнатура метода выглядит следующим образом:

```
public static ParallelLoopResult For(int fromInclusive, int
toExclusive, Action<int> body);
```

Здесь `fromInclusive` – это начальное значение счётчика цикла, `toExclusive` – конечное значение счётчика цикла, `body` – тело цикла (делегат `Action`). В качестве результата, этот метод возвращает структуру, содержащую следующую информацию:

```
public struct ParallelLoopResult
{
public bool IsCompleted { get; }
public long? LowestBreakIteration { get; }
}
```

`IsCompleted` – указывает, выполнен ли цикл успешно или был прерван; `LowestBreakIteration` – минимальный номер итерации цикла где был вызван оператор `break`.

Чтобы продемонстрировать работу метода `Parallel.For`, вернемся к нашему методу определения простых чисел. Сейчас он выглядит следующим образом:

```
static void Prime(int x)
{
Console.WriteLine($"Запускаем расчёт простых чисел от 2 до {x}.
CurrentId = {Task.CurrentId}");
int total = 0;
for (int i = 2; i <= x; i++)
{
bool isPrime = true;
for (int j = 2; j < i; j++)
{
if (i % j == 0)
{
isPrime = false;
break;
```

```
}}
if (isPrime)
total++;
}
Console.WriteLine($"Расчёт простых чисел выполнен. Найдено
{total} чисел");
}
```

Сделаем следующее:

1. Внешний цикл for будет выполняться параллельно (с использованием метода Parallel.For)

2. Внутренний цикл for будет выводить простые числа в консоль и, по сути, являться телом цикла Parallel.For.

Так будет выглядеть определение простого числа (тело цикла Parallel.For):

```
static public void IsPrime (int x)
{
bool isPrime = true;
for (int i = 2; i < x; i++)
{
if (x % i == 0)
{
isPrime = false;
break;
}}
if (isPrime)
Console.WriteLine(x);
}
```

А вот так будет вызываться метод Parallel.For:

```
Parallel.For(2, 5001, IsPrime);
```

То есть цикл будет стартовать с 2 и заканчиваться на значении 5000 включительно. При этом, наш метод IsPrime принимает в качестве параметра целое число int (номер итерации цикла for). Таким образом, можно легко распараллелить выполнение циклов в нашей программе.

9.3. Метод Parallel.ForEach

Этот метод выполняет цикл foreach в параллельном режиме. Сигнатура метода следующая:

```
public static ParallelLoopResult
ForEach<TSource>(IEnumerable<TSource> source, Action<TSource>
body);
```

Чтобы воспользоваться этим методом, снова вернемся к нашему определению простых чисел. Вот так, например, можно определить простые числа из заданного набора, а не по порядку от 2 до x, как в предыдущем примере:

```
Parallel.ForEach<int>(new List<int> { 10, 12, 2, 3, 4, 5, 6, 7,
89, 198 }, IsPrime);
```

9.4. Выход из цикла

В обычных циклах `for` и `foreach` предусмотрены специальные операторы управления циклом, в частности, оператор `break` для выхода из цикла. При использовании класса `Parallel` у нас также есть возможность выйти из цикла на любой его итерации. Например, выйдем из цикла `foreach` при нахождении первого простого числа в списке `List`.

Перепишем метод `IsPrime` следующим образом:

```
static public void IsPrime (int x, ParallelLoopState
parallelLoopState)
{
bool isPrime = true;
for (int i = 2; i < x; i++)
{
if (x % i == 0)
{
isPrime = false;
break;
}}
if (isPrime)
{
Console.WriteLine($"Первое найденное простое число {x}");
parallelLoopState.Break();
}}
```

Теперь наш метод в качестве второго параметра получает объект `ParallelLoopState`, который позволяет управлять циклом. Как только находим в очередной итерации цикла простое число, то при первом же удобном случае вызывается метод `Break` прекращающий выполнение цикла. При этом, следует обратить внимание на то, что вызов `Break` не означает, что цикл прекратиться немедленно, т.к. в момент вызова `Break` параллельно могут рассчитываться и другие итерации цикла. Например,

попробуем вызвать Parallel.ForEach, используя наш метод IsPrime следующим образом:

```
ParallelLoopResult res = Parallel.ForEach<int>(new List<int> {
10, 12, 2, 3, 4, 5, 6, 7, 89, 198 }, IsPrime);
if (res.IsCompleted)
Console.WriteLine("Цикл выполнен полностью");
Console.WriteLine($"Номер итерации, на которой сработал break -
{res.LowestBreakIteration}");
```

В результате, можно увидеть в консоли следующие строки:

```
Первое найденное простое число 7
Первое найденное простое число 5
Первое найденное простое число 89
Первое найденное простое число 2
Первое найденное простое число 3
Номер итерации, на которой сработал break – 2
```

То есть на втором шаге цикла сработал метод Break, прерывающий выполнение цикла. Однако, прежде чем цикл остановился и получили результат, было выполнено ещё несколько параллельных итераций, которые не было возможности моментально прервать.

Таким образом, класс Parallel библиотеки TPL позволяет достаточно легко и удобно выполнять параллельные задачи в приложении. Метод Invoke позволяет запустить параллельное выполнение массива задач, а методы For и ForEach позволяют организовать параллельное выполнение итераций цикла.

Задания и порядок выполнения работы

Задание 1. Выполните все примеры, представленные в теоретической части материала.

Задание 2. Создайте приложение, которое параллельно выполняет вычисление факториала для нескольких чисел. Каждое вычисление факториала должно выполняться в отдельном задании.

Задание 3. Разработайте приложение, которое параллельно обрабатывает элементы массива и находит сумму квадратов этих элементов с использованием Parallel.ForEach.

Контрольные вопросы

1. Что такое параллельное программирование и в каких случаях оно может быть полезно?
2. Дайте определение библиотеке TPL в C#.
3. В чем заключаются основные принципы работы с TPL?
4. Что такое Task и чем она отличается от других объектов в параллельном программировании?
5. Как создать и запустить задачу с использованием TPL?
6. Какие способы синхронизации задач вы знаете и как их применять?
7. Что такое Dataflow и как он используется в TPL?
8. Опишите основные типы блокировок, используемых в TPL, и их влияние на производительность.
9. В каких случаях следует использовать Parallel и Task.WhenAll, а в каких – Task.WhenAny?
10. Что такое «задача-обертка» и для чего она используется в TPL?

Лабораторная работа № 31

Тема: **«Валидация данных, вводимых пользователем в C#»**.

Цель: изучение и практическое применение методик проверки и обработки некорректных или невалидных данных, вводимых пользователем, в приложениях на языке программирования C#.

Краткие теоретические сведения

Важную роль при разработке приложений играет валидация (проверка) данных, вводимых пользователями. Любая модель, используемая в приложении, содержит определенный набор данных и эти данные должны соответствовать каким-либо критериям. Например, если запрашиваем у пользователя возраст, то ожидаем, что будет введено число из какого-либо диапазона, например, от 15 до 99. Если пользователь введет значение 146, то, минимум, что произойдет – это сохранение некорректного значения, например, в БД. В других случаях некорректный ввод данных в программу может спровоцировать ошибки в программе.

1. Валидация модели в C#.

В C#, как и в любом другом современном языке программирования, любую задачу можно решить несколькими способами. Например, если вернуться к примеру с возрастом пользователя, то можно было бы провести вот такую валидацию модели:

```
using System;
namespace ConsoleApp1
{
public class User
{
public int Age { get; set; }
public string Name { get; set; }
}
internal class Program
{
static void Main(string[] args)
{
User user = new User();
```

```
Console.WriteLine("Имя:");
user.Name = Console.ReadLine();
Console.WriteLine("Возраст: ");
user.Age = Convert.ToInt32(Console.ReadLine());
if ((user.Age < 15) || (user.Age > 100))
{
Console.WriteLine("Введено некорректное значение возраста");
}
else
Console.WriteLine($"Добро пожаловать, {user.Name}");
}}}
```

Используя конструкцию `if...else`, проверяем возраст пользователя и, если возраст оказывается менее 15 или более 100 лет, то пользователю возвращается сообщение об ошибке, иначе – приветствие. Такая проверка вполне работоспособна, но имеет свои недостатки и самый главный из них – неуниверсальность такого подхода. Что, например, будем делать, если валидацию необходимо провести не одного, а, скажем, десяти свойств и по разным критериям? Конечно, можем воспользоваться всё тем же старым и неуниверсальным подходом с использование `if..else` и написать, в итоге, совершенно не читаемый, но, возможно, рабочий код. А можем использовать те инструменты и возможности валидации модели в C#, которые уже есть в языке программирования.

2. Использование атрибутов для валидации модели

С атрибутами в C# мы уже сталкивались, когда рассматривали тему работы с JSON. Для того, чтобы использовать атрибуты валидации модели в C# необходимо в проект подключить пространство имен `System.ComponentModel.DataAnnotations`. В этом пространстве имен содержатся различный классы атрибутов для валидации данных, некоторыми из которых мы сейчас воспользуется. Перепишем пример с пользователем следующим образом:

```
using System;
using System.Collections.Generic;
using System.ComponentModel.DataAnnotations;
namespace ConsoleApp1
{
public class User
```

```
{
[Required]
[Range(15,100)]
public int Age { get; set; }
[Required]
[StringLength(50, MinimumLength = 3)]
public string Name { get; set; }
}
internal class Program
{
static void Main(string[] args)
{
User user = new User();
Console.WriteLine("Имя:");
user.Name = Console.ReadLine();
Console.WriteLine("Возраст: ");
user.Age = Convert.ToInt32(Console.ReadLine());
var results = new List<ValidationResult>();
var context = new ValidationContext(user);
if (!Validator.TryValidateObject(user, context, results, true))
{
foreach (var error in results)
{
Console.WriteLine(error.ErrorMessage);
}}}}}
```

Посмотрим, что изменилось в приложении. Во-первых, для свойств класса `User` определили по два атрибута:

1. `Required` – означает, что свойство является обязательным для определения

2. `Range` – означает, что число, введенное пользователем, должно находиться в заданном диапазоне (в нашем случае – от 15 до 100)

3. `StringLength` – определяет длину строки (в нашем случае, строка с именем должна быть не менее трех и не более пятидесяти символов)

В методе `Main` использовали три класса из пространства имен `System.ComponentModel.DataAnnotations`: `ValidationResult`, `Validator` и `ValidationContext`.

Используя класс `ValidationContext`, мы создаем так называемый контекст валидации и в конструкторе, в первом параметре, передаем объект, который необходимо проверить. В нашем случае, это объект класса `User`. Далее, передаем этот

контекст в метод TryValidateObject класса Validator. Метод возвращает значение true, если проверка объекта прошла успешно и все свойства заполнены корректно. Если же при валидации модели обнаружены какие-либо ошибки, то метод вернет значение false, а обнаруженные ошибки будут сохранены в списке List<ValidationResult>.

Результат работы программы в случае обнаружения ошибок может быть следующим:

```
Имя:
Ю
Возраст:
12
The field Age must be between 15 and 100.
The field Name must be a string with a minimum length of 3 and
a maximum length of 50.
```

Таким образом, вместо нагромождения различных конструкций типа if...else в коде, мы воспользовались классами валидации и проверили нашу модель пользователя на корректность введенных данных. Код приложения стал понятнее, а его поддержка – проще, так как все требования к модели описали с помощью атрибутов валидации в самом классе User.

Таким образом, рассмотрели основные моменты валидации модели в C# с использованием атрибутов валидации. Для валидации могут использоваться три класса ValidationResult, Validator и ValidationContext из пространства имен System.ComponentModel.DataAnnotations.

3. Атрибуты валидации

Рассмотрим, как конкретизировать правила валидации определенных полей и свойств объекта с использованием атрибутов.

Все атрибуты валидации, которые будем рассматривать ниже, являются потомками класса ValidationAttribute. У этого класса определены основные свойства, необходимые для настройки валидации и одно из главных свойств – это свойство ErrorMessage, которое содержит текст ошибки. Используя это свойство, можно локализовать текст ошибки, который увидит пользователь. Например, перепишем класс User следующим образом:

```
public class User
{
[Required]
[Range(15,100, ErrorMessage ="Возраст пользователя должен быть
не менее 15 и не более 100 лет")]
public int Age { get; set; }
[Required]
[StringLength(50, MinimumLength = 3)]
public string Name { get; set; }
}
```

Теперь запустим приложение и посмотрим на результат валидации модели, если будет задан некорректный возраст:

```
Имя:
Вася
Возраст:
11
Возраст пользователя должен быть не менее 15 и не более 100 лет
```

4. Основные атрибуты валидации моделей

В пространстве имен `System.ComponentModel.DataAnnotations` содержится большое количество атрибутов валидации, покрывающих, в принципе, большинство наших потребностей. Рассмотрим основные из них.

4.1. Атрибут Required

Этот атрибут означает, что поле (или свойство) модели должно быть обязательно заполнено. При этом, ошибка валидации возникает в следующих случаях:

– если свойство имеет значение `null`

– содержит пустую строку (`""`)

– содержит только символы пробела.

Если для поля допускается ввод пустой строки, то можно определить атрибут `Required` со следующим свойством:

```
public class User
{
<--Прочие свойства-->
[Required(AllowEmptyStrings = true)]
public string Comment { get; set; } = null;
}
```

Если допускается, что поле или свойство может содержать пустую строку или строку, состоящую из пробелов, то при значении null всё равно будет возникать ошибка валидации.

4.2. Атрибут StringLength

Атрибут используется для указания минимальной и максимальной длину строки, разрешенной в поле данных:

```
public class User
{
<--Прочие свойства класса-->
[StringLength(50, MinimumLength = 3)]
public string Name { get; set; }
}
```

В первом параметре задали максимальную длину строки, а во втором – минимальную.

4.3. Атрибут Range

Атрибут Range задает ограничения числового диапазона для значения поля данных. Ранее использовали этот атрибут для проверки возраста пользователя:

```
public class User
{
[Range(15,100, ErrorMessage ="Возраст пользователя должен быть
не менее 15 и не более 100 лет")]
public int Age { get; set; }
<--Прочие свойства класса-->
}
```

При этом, атрибут Range может применяться и к полям с типом данных DateTime:

```
[Range(typeof(DateTime),"01/01/2006", "01/01/2121",
ErrorMessage ="Введено некорректное значение даты")]
public DateTime Birthday { get; set; }
```

4.4. Атрибут RegularExpression

Атрибут RegularExpression указывает на то, что значение должно соответствовать определенному в параметрах атрибута регулярному выражению. Этот атрибут удобно использовать,

например, для проверки адреса электронной почты или телефона пользователя:

```
[RegularExpression(@"^([a-z0-9_-]+\.)*[a-z0-9_-]+@[a-z0-9_-
]+(\.[a-z0-9_-]+)*\.[a-z]{2,6}$")]
public string Email { get; set; }
```

В случае, если пользователь введет некорректное значение e-mail, то программа вернет ошибку с текстом, сообщающим о том, что значение поля `Email` не соответствует регулярному выражению. При необходимости, текст этого сообщения можно изменить с использованием свойства `ErrorMessage`.

Для проверки адреса электронной почты не обязательно использовать регулярное выражение. В C# для этого также имеется специальный атрибут – `EmailAddress`.

4.5. Атрибут Compare

Атрибут `Compare` позволяет сравнивать два свойства объекта. Например, этот атрибут удобно использовать для сравнения паролей, которые вводит пользователь:

```
public class User
{
<--Прочие свойства класса-->
public string Password { get; set; }
[Compare("Password")]
public string ConfirmPassword { get; set; }
}
```

В параметре атрибута передаем имя свойства, с которым будем сравнивать значение текущего свойства. В данном случае, значение свойства `ConfirmPassword` сравнивалось со значением свойства `Password`.

4.6. Атрибут Phone

Атрибут `Phone` указывает на то, что поле содержит номер телефона в правильном формате. По сути, атрибуты `Phone` и `EmailAddress` – это встроенные атрибуты `RegularExpression`.

4.7. Атрибут CreditCard

Атрибут `CreditCard` указывает на то, что значение поля данных – это номер кредитной карты. Этот атрибут не проверяет

доступность карты для покупок, а только проверяет соответствие строки заданному формату.

4.8. Атрибут Url

Атрибут `Url` проверяет является ли введенное значение `url`-адресом. Как и в случае с атрибутом `CreditCard`, проверка на доступность введенного адреса не производится.

Таким образом, наличие различных атрибутов валидации модели в C# обеспечивает большинство наших потребностей при проверке введенных пользователем данных на корректность.

5. Создание собственных атрибутов валидации

Валидация модели в C# не ограничивается только этими атрибутами – нам может потребоваться более сложная проверка, для которой готовых атрибутов валидации может не хватить. В этом случае можем создать собственный атрибут валидации и использовать его в своем приложении.

5.1. Создание атрибута валидации свойства модели

Напишем свой собственный атрибут, который будет проверять вхождение заданного числа не в один, а два диапазона.

```
class RangeExtAttribute : ValidationAttribute
{
public int MinimumFirst { get; set; }
public int MaximumFirst { get; set; }
public int MinimumLast { get; set; }
public int MaximumLast { get; set; }
public RangeExtAttribute(int minimumFirst, int maximumFirst,
int minimumLast, int maximumLast)
{
MinimumFirst = minimumFirst;
MaximumFirst = maximumFirst;
MinimumLast = minimumLast;
MaximumLast = maximumLast;
}
public override bool IsValid(object? value)
{
if (value == null)
{
ErrorMessage = "Значение свойства не должно быть равным null";
return false;
```

```
}
if (value.GetType() != typeof(int))
{
ErrorMessage = "Свойства должно должно иметь тип int";
return false;
}
int intValue = (int)value;
if ((intValue < MinimumFirst) || (intValue > MaximumLast) ||
((intValue > MaximumFirst) && (intValue < MinimumLast)))
{
ErrorMessage = $"Значение свойства должно лежать в диапазоне
[{MinimumFirst},{MaximumFirst}]      или      в      диапазоне
[{MinimumLast},{MaximumLast}]";
return false;
}
return true;
}}
```

Часть необходимых проверок (например, на пересечение диапазонов) опущена в целях сокращения кода.

Для создания собственного атрибута валидации мы создали класс-наследник от `ValidationAttribete` и переопределили в нем метод `IsValid`. Этот метод должен возвращать `true`, если свойство прошло валидацию и `false` – в ином случае. Так как проверяем только значения `int`, то вначале производится проверка проверяемого значения на `null` и тип данных. Далее –проверяем число на вхождение в один из заданных диапазонов и, если проверка прошла успешно, то возвращаем значение `true`.

Воспользоваться этим атрибутом также просто, как и уже имеющимися атрибутами валидации из пространства имен `System.ComponentModel.DataAnnotations`

```
class Model
{ [RangeExt(0,5,7,10)]
public int Value { get; set; }}
```

Обратите внимание, что можно опустить в имени класса атрибута часть `Attribute`. Проверим работу атрибута:

```
Введите целое число в диапазоне [0,5] или [7,10] 6
Значение свойства должно лежать в диапазоне [0,5] или в
диапазоне
[7,10]
```

5.2. Создание атрибута валидации для модели в целом

Атрибуты валидации, применяемые сразу ко всей модели, чаще всего, используются для проверки нескольких свойств. В целом же, создание такого атрибута ничем не сложнее, чем в предыдущем примере. Например:

```
public class AllRequiredAttribute: ValidationAttribute
{
public override bool IsValid(object? value)
{
Model model = value as Model;
if (model.Value == 0)
{
ErrorMessage = "Значение Value не может быть равно нулю";
return false;
}
if (model.Name == null)
{
ErrorMessage = "Значение Name не может быть равно null";
return false;
}
return true;
}}
```

Производим проверку модели, у которой значение `Value` не должно быть равным нулю, а свойство `Name` не должно быть равно `null`. Проверим работу атрибута следующим образом:

```
[AllRequired]
class Model
{
[RangeExt(0,5,7,10)]
public int Value { get; set; }
public string? Name { get; set; } = null;
}
```

Результат работы программы:

```
Введите целое число в диапазоне [0,5] или [7,10] 5
Значение Name не может быть равно null
```

Собственные атрибуты валидации должны являться наследниками класса `ValidationAttribute` и переопределять метод `IsValid`. В зависимости от наших потребностей мы можем создавать как атрибуты для проверки отдельных свойств модели, так и модели (объекта) целиком.

6. **Интерфейс** IValidatableObject

Для валидации модели в C# не обязательно создавать свои собственные атрибуты валидации. Разрабатываемая модель (класс) может наследовать интерфейс `IValidatableObject` и валидация свойств модели может проводиться внутри метода `Validate`. Рассмотрим пример реализации интерфейса `IValidatableObject` в нашем приложении.

Реализуем интерфейс `IValidatableObject` на примере класса пользователя, с которым работали ранее.

```
public class User: IValidatableObject
{
public int Age { get; set; }
public string Name { get; set; }
public string Password { get; set; }
public string ConfirmPassword { get; set; }
public                              IEnumerable<ValidationResult>
Validate(ValidationContext validationContext)
{
List<ValidationResult> results = new List<ValidationResult>();
if ((Name == null) || (Name.Trim() == ""))
results.Add(new     ValidationResult("Не     определено     имя
пользователя"));
if ((Age<15)||(Age>100))
results.Add(new ValidationResult("Возраст пользователя должен
быть от 15 до 100 лет"));
if (Password != ConfirmPassword)
{
results.Add(new ValidationResult("Пароли не совпадают"));
}
return results;
}}
```

Здесь мы, фактически, заменили все атрибуты валидации на один метод – `Validate` интерфейса `IValidatableObject`. Теперь можем воспользоваться классом `Validate` и провести валидацию пользователя, например, так:

```
internal class Program
{
static void Main(string[] args)
{
User user = new User();
Console.WriteLine("Имя:");
user.Name = Console.ReadLine();
Console.WriteLine("Возраст: ");
```

```
user.Age = Convert.ToInt32(Console.ReadLine());
Console.WriteLine("Введите пароль: ");
user.Password = Console.ReadLine();
Console.WriteLine("Подтвердите пароль: ");
user.ConfirmPassword = Console.ReadLine();
var results = new List<ValidationResult>();
var context = new ValidationContext(user);
if (!Validator.TryValidateObject(user, context, results, true))
{
foreach (var error in results)
{
Console.WriteLine(error.ErrorMessage);
}}}}
```

Результат работы программы:

```
Имя:
Вася
Возраст:
14
Введите пароль:
1234567890
Подтвердите пароль:
0987654321
Возраст пользователя должен быть от 15 до 100 лет
Пароли не совпадают
```

Таким образом, реализуя интерфейс `IValidatableObject`, получаем фактически возможность т.н. самовалидации модели – вся проверка соответствия полей и свойств класса осуществляется непосредственно в методе `Validate`.

Задания и порядок выполнения работы

Задание 1. Выполнить все примеры, приведенные в теоретическом материале. По каждому из примеров создать скриншот с окном Visual Studio в режиме запуска приложения с примером. Первой строкой, которая выводится в консоль должна быть строка со своим ФИО и текущей датой.

Задание 2. Используя валидацию данных, написать программу, которая проверяет, является ли введенное пользователем число четным или нечетным.

Задание 3. Используя валидацию данных, написать программу, проверяющую, является ли введенный пользователем email-адрес действительным.

Контрольные вопросы

1. Что такое валидация данных и почему она важна?
2. Какие существуют типы валидации данных?
3. Каковы основные этапы процесса валидации данных в C#?
4. Как проверить, является ли введенная строка числом, датой или email-адресом?
5. Какие регулярные выражения можно использовать для проверки правильности ввода?
6. Как обрабатывать исключения, возникающие при некорректном вводе?
7. В чем разница между локальной и серверной валидацией данных?
8. Как обеспечить безопасность данных, полученных от пользователя?
9. Как интегрировать валидацию данных с формами HTML и ASP.NET MVC?
10. Какие инструменты и библиотеки можно использовать для валидации данных на C#?

Лабораторная работа № 32

Тема: «Основы работы с базами данных в C#. СУБД SQLite. Библиотека SQLite.Net»

Цель: изучить основы работы с базами данных на примере работы с встраиваемой СУБД SQLite на языке программирования C#, а также с использованием библиотеки SQLite.NET.

Краткие теоретические сведения

SQLite – это встраиваемая СУБД, когда система управления встраивается в саму программу. Это значит, что все запросы и команды идут в базу не через посредника, а напрямую из приложения. Чтобы встроить SQLite в код, достаточно подключить нужную библиотеку.

Все данные в SQLite хранятся в одном файле – таблицы, служебные поля, связи и всё остальное. Это упрощает работу с базой и позволяет легко переносить данные из одного места в другое.

1. Установка необходимых пакетов

Для того, чтобы иметь возможность работать с БД SQLite в C# нам потребуется пакет System.Data.SQLite. Чтобы его установить, достаточно воспользоваться менеджером пакетов NuGet. Выбираем в контекстном меню проекта «Управление пакетами NuGet...» (рис. 32.1).

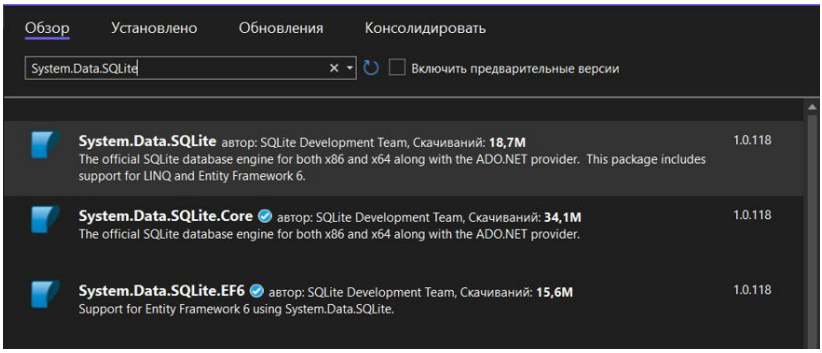

Рис. 32.1. Установка необходимых пакетов

Нажимаем OK (рис. 32.2).

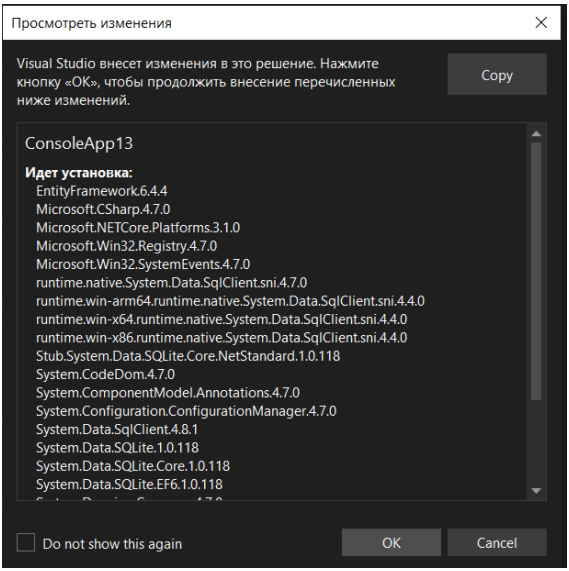

Рис. 32.2. Конфигурация установки

Соглашаемся с условиями (рис 32.3).

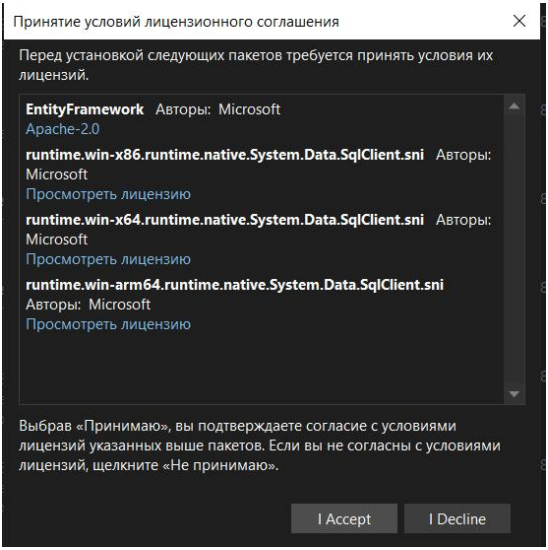

Рис. 32.3. Окно лицензионного соглашения

Смотрим зависимости (рис. 32.4)

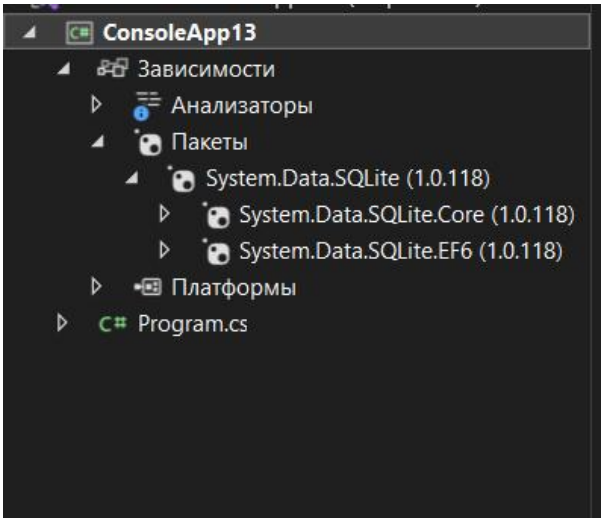

Рис. 32.4. Зависимости проекта

2. Подключение к БД SQLite в C#

Создадим консольное приложение со следующим кодом

```
using System.Data.SQLite;
namespace SQLiteExample
{
    class Program
    {
        static SQLiteConnection connection;
        static SQLiteCommand command;

        static public bool Connect(string fileName)
        {
            try
            {
                connection = new SQLiteConnection("Data
Source=" + fileName + ";Version=3; FailIfMissing=False");
                connection.Open();
                return true;
            }
            catch (SQLiteException ex)
            {
```

```
                Console.WriteLine($"Ошибка доступа к базе
данных. Исключение: {ex.Message}");
                return false;
            }
        }

        static void Main(string[] args)
        {
            if (Connect("firstBase.sqlite"))
            {
                Console.WriteLine("Connected");
            }
        }
    }
}
```

Пробуем подключиться к базе данных firstBase.sqlite (рис. 32.5), используя свой метод `Connect`. Метод `Connect()` получает путь к файлу базы данных (`fileName`), создает строку подключения к базе данных и пробует подключиться к этой базе данных. Если подключение прошло успешно, то метод возвращает `true`.

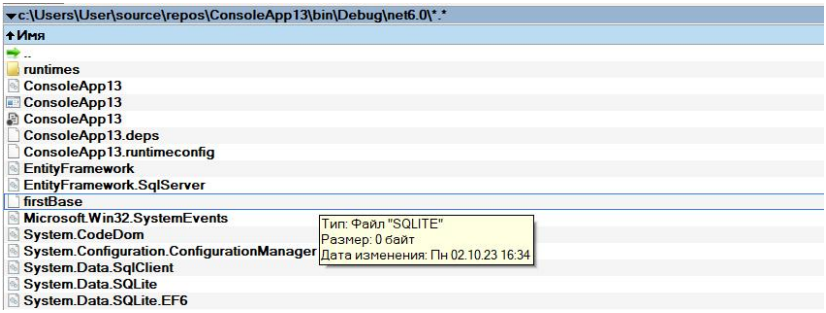

Рис. 32.5. Каталог с файлами проекта

Строка подключения состоит из следующих элементов:

`Data Source` – источник (файл базы данных), к которому необходимо подключиться;

`Version` – версия БД (в нашем случае, этот параметр равен 3)

`FailIfMissing` – параметр, определяющий действие в случае, если файл базы данных не будет найден при подключении:

true – сгенерировать исключение, если файл БД не будет найден; false – файл БД будет автоматически создан (значение по умолчанию). Так, если установить этому параметру значение true и попробовать подключиться к несуществующему файлу БД SQLite, то получим следующее исключение:

```
Ошибка доступа к базе данных. Исключение: unable to open
database file
```

3. Создание таблиц БД SQLite в C#

Изменим содержимое метода Main добавив команду для создания таблицы

```
static void Main(string[] args)
        {
            if (Connect("firstBase.sqlite"))
            {
                Console.WriteLine("Connected");
                command = new SQLiteCommand(connection)
                {
                    CommandText = "CREATE TABLE IF NOT EXISTS
[Person]([id] INTEGER PRIMARY KEY AUTOINCREMENT NOT NULL UNIQUE,
[name] TEXT, [family] TEXT, [age] byte);"
                };
                command.ExecuteNonQuery();
                Console.WriteLine("Таблица создана");
            }
        }
```

Для того, чтобы выполнять команды к базе данных SQLite в C#, мы создали новый объект типа SQLiteCommand, определили текст команды и вызвали метод ExecuteNonQuery(). Метод ExecuteNonQuery в результате возвращает количество строк затронуты выполнением запроса. Теперь у нас есть файл базы данных SQLite, в котором имеется таблица Person.

4. Запись данных в БД SQLite в C#

Добавим одну запись в таблицу базы данных SQLite, используя все тот же объект типа SQLiteCommand:

```
command.CommandText = "INSERT INTO Person (name, family, age)
VALUES (\"Иванов\",\"Иван\", 25)";
command.ExecuteNonQuery();
```

В приведенном выше примере использовали в качестве запроса обычную строку с заранее известными параметрами

запроса. Однако, удобнее использовать для добавления новых записей в БД SQLite запрос с параметрами. Сделать это можно, например, следующим образом:

```
command.CommandText = "INSERT INTO Person (name, family, age)
VALUES (:name, :family, :age)";
command.Parameters.AddWithValue("name", "Сергей");
command.Parameters.AddWithValue("family", "Петров");
command.Parameters.AddWithValue("age", 13);
command.ExecuteNonQuery();
```

5. Использование транзакций SQLite

Одной из претензий, которую предъявляли новички в работе с SQLite к этой базе данных является то, что, по мнению новичков, запись в эту базу данных происходит очень медленно. Действительно, посмотрим на вот такой код:

```
using System.Data;
using System.Data.SQLite;
using System.Diagnostics;


namespace SQLiteExample
{
    class Program
    {
        static SQLiteConnection connection;
        static SQLiteCommand command;

        static public bool Connect(string fileName)
        {
            try
            {
                connection = new SQLiteConnection("Data
Source=" + fileName + ";Version=3; FailIfMissing=False");
                connection.Open();
                return true;
            }
            catch (SQLiteException ex)
            {
                Console.WriteLine($"Ошибка доступа к базе
данных. Исключение: {ex.Message}");
                return false;
```

```
            }
        }

        static void Main(string[] args)
        {
            if (Connect("firstBase.sqlite"))
            {
                Console.WriteLine("Connected");
                command = new SQLiteCommand(connection)
                {
                    CommandText = "CREATE TABLE IF NOT EXISTS
[Person]([id] INTEGER PRIMARY KEY AUTOINCREMENT NOT NULL UNIQUE,
[name] TEXT, [family] TEXT, [age] byte);"
                };
                command.ExecuteNonQuery();
                Console.WriteLine("Таблица создана");
            }
            if (Connect("firstBase.sqlite"))
            {
                Console.WriteLine("Connected");
                Stopwatch sw = new Stopwatch();
                sw.Restart();
                command.CommandText  =  "INSERT  INTO  Person
(name, family, age) VALUES (:name, :family, :age)";
                try
                {
                    for (int i = 1; i < 1001; i++)
                    {

command.Parameters.AddWithValue("name", "Сергей");

command.Parameters.AddWithValue("family", "Петров");
                        command.Parameters.AddWithValue("age",
i);
                        command.ExecuteNonQuery();
                    }
                    sw.Stop();
                    Console.WriteLine(sw.Elapsed);
                }
                catch
                {
                    throw;
```

```
                }

            }
        }
    }
}
```

Пробуем записать в базу SQLite 1000 записей и, при этом, засекаем время выполнения операции (необходимо добавить пространство имен System.Diagnostics). Вот, что покажет нам счётчик времени выполнения операции:

```
00:00:09.6012235
```

Девять секунд на добавление 1000 записей в базу данных – это достаточно много. Ускорить время выполнения операции добавления записей SQLite на порядок позволяет использование транзакций. Перепишем наш код следующим образом:

```
using System.Data;
using System.Data.Common;
using System.Data.SQLite;
using System.Diagnostics;
using System.Transactions;


namespace SQLiteExample
{
    class Program
    {
        static SQLiteConnection connection;
        static SQLiteCommand command;

        static public bool Connect(string fileName)
        {
            try
            {
                connection = new SQLiteConnection("Data
Source=" + fileName + ";Version=3; FailIfMissing=False");
                connection.Open();

                return true;
            }
            catch (SQLiteException ex)
            {
```

```
                Console.WriteLine($"Ошибка доступа к базе
данных. Исключение: {ex.Message}");
                return false;
            }
        }

        static void Main(string[] args)
        {
            if (Connect("firstBase.sqlite"))
            {
                Console.WriteLine("Connected");
                command = new SQLiteCommand(connection)
                {
                    CommandText = "CREATE TABLE IF NOT EXISTS
[Person]([id] INTEGER PRIMARY KEY AUTOINCREMENT NOT NULL UNIQUE,
[name] TEXT, [family] TEXT, [age] byte);"
                };
                command.ExecuteNonQuery();
                Console.WriteLine("Таблица создана");
            }

            command.CommandText = "INSERT INTO Person (name,
family, age) VALUES (:name, :family, :age)";
             connection.BeginTransaction();//запускаем
транзакцию
            try
            {
                for (int i = 1; i < 1001; i++)
                {
                    command.Parameters.AddWithValue("name",
"Сергей");
                    command.Parameters.AddWithValue("family",
"Петров");
                    command.Parameters.AddWithValue("age", i);
                    command.ExecuteNonQuery();
                }
                Stopwatch sw = new Stopwatch();
                sw.Restart();

                connection.BeginTransaction().Commit();
//применяем изменения
                sw.Stop();
```

```
                Console.WriteLine(sw.Elapsed);
            }
            catch
            {
                connection.BeginTransaction().Rollback();
//откатываем изменения, если произошла ошибка
                throw;
            }
        }
    }
}
```

Посмотрим, что покажет счётчик времени выполнения операции:

```
00:00:00.0010033
```

Как видите, выполнение транзакции в SQLite позволило сократить время выполнения операции более, чем на 99%.

6. Чтение данных из БД SQLite в C#

Для того, чтобы понять как происходит чтение данных из таблиц БД SQLite в C# можно воспользоваться следующим простым примером:

```
using System.Data;
using System.Data.Common;
using System.Data.SQLite;
using System.Diagnostics;
using System.Transactions;


namespace SQLiteExample
{
    class Program
    {
        static SQLiteConnection connection;
        static SQLiteCommand command;

        static public bool Connect(string fileName)
        {
            try
            {
                connection   =   new   SQLiteConnection("Data
Source=" + fileName + ";Version=3; FailIfMissing=False");
                connection.Open();
```

```
                return true;
            }
            catch (SQLiteException ex)
            {
                Console.WriteLine($"Ошибка доступа к базе
данных. Исключение: {ex.Message}");
                return false;
            }
        }

        static void Main(string[] args)
        {
            if (Connect("firstBase.sqlite"))
            {
                Console.WriteLine("Connected");
                command = new SQLiteCommand(connection)
                {
                    CommandText = "CREATE TABLE IF NOT EXISTS
[Person]([id] INTEGER PRIMARY KEY AUTOINCREMENT NOT NULL UNIQUE,
[name] TEXT, [family] TEXT, [age] byte);"
                };
                command.ExecuteNonQuery();
                Console.WriteLine("Таблица создана");
                command.CommandText = "INSERT INTO Person
(name, family, age) VALUES (:name, :family, :age)";
                command.Parameters.AddWithValue("name",
"Сергей");
                command.Parameters.AddWithValue("family",
"Петров");
                command.Parameters.AddWithValue("age", 13);
                command.ExecuteNonQuery();


                command.CommandText = "SELECT * FROM Person";
                DataTable data = new DataTable();
                SQLiteDataAdapter adapter = new
SQLiteDataAdapter(command);
                adapter.Fill(data);
                Console.WriteLine($"Прочитано
{data.Rows.Count} записей из таблицы БД");
                foreach (DataRow row in data.Rows)
                {
                    Console.WriteLine($"id =
{row.Field<long>("id")} name = {row.Field<string>("name")}
family = {row.Field<string>("family")}");
                }
            }
```

```
        }
    }
}
```

Объект типа DataTable представляет собой таблицу данных в памяти. В этой таблице (как и в любой другой таблице) все данные хранятся в виде столбцов и строк, где каждая строка – это отдельная запись таблицы БД. В свою очередь объект типа SQLiteDataAdapter предоставляет нам набор команд и методов для работы с наборами данных.

Для чтения данных из БД SQlite вначале задаем текст команды, которую необходимо выполнить в БД – это команда SELECT, с помощью которой считываем все данные из таблицы Person. Далее создаем SQLiteDataAdapter, используя для этого один из конструкторов, в который передаем команду (объект SQLiteCommand) и таблицу (DataTable) для хранения полученных результатов запроса (рис. 32.6).

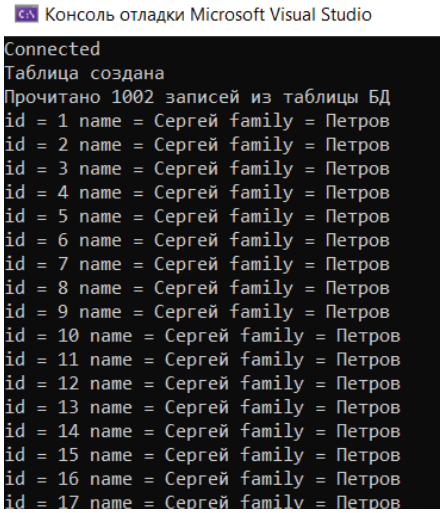

Рис. 32.6. Результат выполнения в консоли

У SQLiteDataAdapter есть несколько конструкторов, позволяющих создать объект. Мы воспользовались только одним из них.

После этого заполняем нашу таблицу данными, используя для этого метод Fill() адаптера. Как только заполнили данными нашу таблицу, можем приступать к чтению данных по каждой записи. Для этого в примере использован цикл foreach в котором мы проходимся по каждой строке таблицы и в каждой строке читаем поля id, name и family каждой записи, используя обобщенный метод Field<T>(), который позволяет привести данные поля к необходимому нам типу.

Обратите внимание на чтение поля id – здесь приводим значение поля к типу long, а не int, как могло бы показаться более правильным. В таблице SQLite это поле определено как уникальный идентификатор типа INTEGER, что в C# соответствует типу long. Если вы попробуете привести значение этого поля к int, то получите исключение:

```
System.InvalidCastException HResult=0x80004002 Сообщение =
Unable to cast object of type 'System.Int64' to type
'System.Int32'. Источник = System.Private.CoreLib
```

Если файл базы данных удалить или переименовать, то результат уже будет только с одной записью (рис. 32.7).

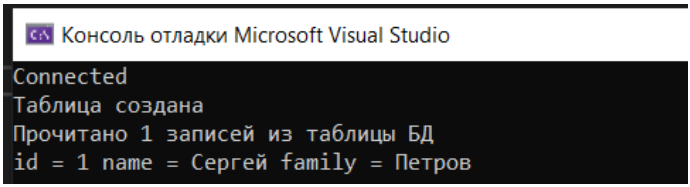

Рис. 32.7. Результат выполнения команд

7. Использование библиотеки SQLite.Net

SQLite.NET – это небольшая библиотека для работы с базами данных SQLite в C#, разработанная с использованием ORM-технологии (object-relational mapping – отображения данных на реальные объекты), которая позволяет абстрагироваться от непосредственного создания базы данных и оперировать всеми операциями в базе данных на уровне объектов C#.

Добавление SQLite.NET в проект. В строке поиска запишем sqlite-pcl-net (рис. 32.8)

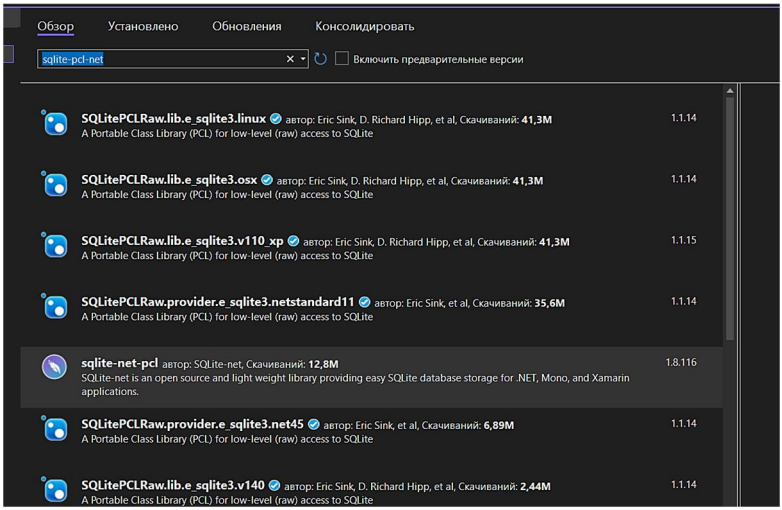

Рис. 32.8. Установка пакетов

Также необходимо согласиться с установкой зависимостей пакета (рис. 32.9).

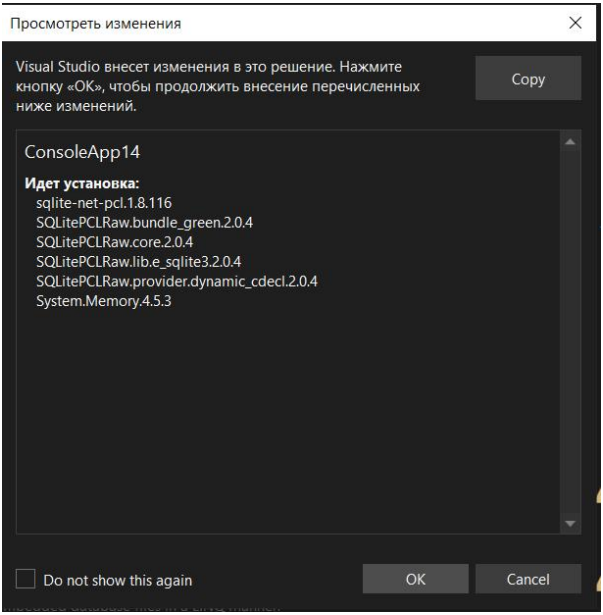

Рис. 32.9. Зависимости пакетов

8. Первое приложение с SQLite.NET

Как было сказано выше, изначально SQLite.NET позиционировалась как легкая, быстрая библиотека для работы с SQLite и разработчикам удалось этого добиться. Попробуем создать свое первое приложение для работы с SQLite. Пусть, например, это приложение работает с данными групп студентов. Для начала, создадим необходимые для работы классы C#. Во-первых, это класс группы:

```
public class Group
{
/// <summary>
/// Идентификатор группы в БД
/// </summary>
public int Id { get; set; }
/// <summary>
/// Название (номер) группы
/// </summary>
public string Name { get; set; }
}
```

Во-вторых, это класс, описывающий одного студента:

```
public class Student
{
/// <summary>
/// Идентификатор студента в БД
/// </summary>
public int Id { get; set; }
/// <summary>
/// Идентификатор группы студента
/// </summary>
public int GroupId { get; set; }
/// <summary>
/// Ф.И.О. студента
/// </summary>
public string Name { get; set; }
}
```

В дальнейшем расширим возможности работы нашего приложения и, соответственно, добавим в классы новые поля и методы, в пока будет достаточно того, что представлено выше.

9. Подключение к БД SQLite и создание необходимых таблиц

Использование технологии ORM позволяет нам выполнять какие-либо операции с БД, оперируя понятиями объектно-ориентированного программирования, то есть – классами, объектами, их свойствами и методами. Так, для этих целей в SQLite.NET используются специальные атрибуты, находящиеся в пространстве имен SQLite. С помощью этих атрибутов мы как бы «размечаем» наши классы, указывая библиотеке, что является таблицей, какие свойства применять к тому или иному столбцу и так далее.

Итак, нам необходимо создать в базе SQLite две таблицы – таблицу, содержащую группы студентов и таблицу, содержащую списки студентов в группах. Подключаем через using пространство имен SQLite и указываем необходимые атрибуты в разработанных выше классах:

```
[Table("groups")]
public class Group
{
[PrimaryKey, AutoIncrement]
[Column("id")]
public int Id { get; set; }
[Column("name")]
public string Name { get; set; }
}
[Table("students")]
public class Student
{
[PrimaryKey, AutoIncrement]
[Column("studentId")]
public int Id { get; set; }
[Column("groupId")]
public int GroupId { get; set; }
[Column("name")]
public string Name { get; set; }
}
```

Рассмотрим, что за атрибуты мы указали.

`Table` – атрибут, применяемый только к классам. Используется для указания того, что представленный `класс` будет отображен в базе данных как таблица. Соответственно, поля

класса будут формировать столбцы таблицы. Атрибут имеет следующее описание:

```
[Table(Name, WithoutRowId)]
```

`Name` – имя таблицы в БД (обязательный параметр)

`WithoutRowId` – указывает на то, что таблица должна создаваться без rowid. По умолчанию этот параметр равен false.

`PrimaryKey` – атрибут, указывающий на то, что свойство является первичным ключом.

`AutoIncrement` – указывает на то, что в БД это свойство класса будет увеличиваться на 1 автоматически (автоинкрементное поле)

`Column` – атрибут, задающий свойства конкретного столбца таблицы. С помощью этого атрибута можно переопределить название столбца в БД. Так, например, свойство Id класса Student будет определено в таблице SQLite как studentId. Атрибут имеет следующее описание:

```
[Column(Name)]
```

`Name` – название столбца в базе данных (обязательный параметр).

Теперь создадим нашу первую базу данных SQLite и посмотрим на результат.

Первый вариант создания таблиц:

```
SQLiteConnection connection = new
SQLiteConnection("students.sqlite");//создаем подключение к БД
//создаем необходимые таблицы
connection.CreateTable<Student>();
connection.CreateTable<Group>();
```

В начале создали подключение к базе данных (SQLiteConnection), которое в дальнейшем будет использоваться для работы с базой данных. После этого, создали две таблицы в новой БД. В данном случае, используя метод `CreateTable`, создали две таблицы.

Второй вариант – использовать метод `CreateTables`, передав в качестве второго параметра массив типов:

```
connection.CreateTables(CreateFlags.None, new Type[] {
typeof(Group), typeof(Student) });
```

В обоих случаях будет создана новая база данных в файле `students.sqlite`, содержащая две таблицы:

```
CREATE TABLE "groups" (
"id" integer primary key autoincrement not null ,
"name" varchar );
CREATE TABLE "students" (
"studentId" integer primary key autoincrement not null ,
"groupId" integer ,
"name" varchar );
```

В приведенном выше примере мы воспользовались самым простым конструктором для SQLiteConnection, просто указав файл БД. При подключении библиотека сама определила есть ли такой файл, создала его и предоставила БД для дальнейшей работы. Так, файл БД создан рядом с exe-файлом приложения (рис. 32.10):

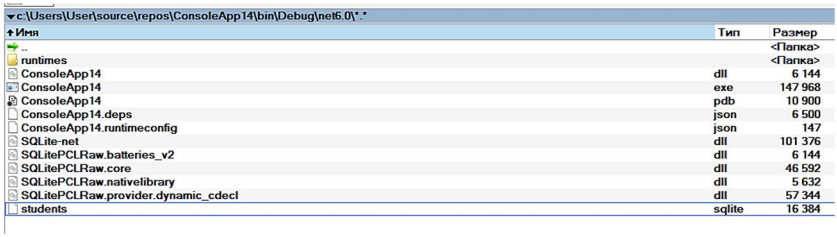

Рис. 32.10. Файл базы данных SQLite

10. Запись и чтение данных

Для первого знакомства с SQLite.NET добавим в наше приложение возможность записывать и читать данные из БД:

```
//добавим в таблицу groups новую группу
connection.Insert(new Group() { Name = "Test" });
//получаем rowId последней добавленной записи в таблицу groups
int lastId = connection.ExecuteScalar<int>("SELECT
last_insert_rowid()");
//добавим в группу нового студента
connection.Insert(new Student() {Name = "Иванов Иван Иванович",
GroupId = lastId });
//считываем данные из таблиц
var groups = connection.Table<Group>();
foreach (Group db_group in groups)
{
Console.WriteLine($"{db_group.Id} - {db_group.Name}");
}
var students = connection.Table<Student>();
foreach (Student db_student in students)
{
Console.WriteLine($"{db_student.Id} - {db_student.Name}");
}
```

Чтобы добавить новые данные в какую-либо таблицу SQLite используем метод Insert, в параметры которого передается объект определенного типа. SQLite.NET сама по типу объекта определяет в какую таблицу его записать. Более того, обратите внимание на то, что мы не указывали значения Id у создаваемых объектов, так как эти свойства в базе данных созданы как autoincrement.

Соответственно, для чтения данных из БД использовали универсальный метод Table, указав тип объектов, которые необходимо вернуть в результате. В результате выполнения метода Table возвращается объект типа TableQuery, который реализует интерфейс IEnumerable, поэтому можно сразу использовать этот объект в цикле foreach для перечисления всех объектов. Результат выполнения представленного выше кода будет следующий:

```
1 – Test
1 – Иванов Иван Иванович
```

Весь код в данном случае принимает вид

```
namespace ConsoleApp14
{
    using SQLite;

    [Table("groups")]
    public class Group
    {
        [PrimaryKey, AutoIncrement]
        [Column("id")]
        public int Id { get; set; }
        [Column("name")]
        public string Name { get; set; }
    }

    [Table("students")]
    public class Student
    {
        [PrimaryKey, AutoIncrement]
        [Column("studentId")]
        public int Id { get; set; }
        [Column("groupId")]
        public int GroupId { get; set; }
        [Column("name")]
        public string Name { get; set; }
```

```
    }
    internal class Program
    {
        static void Main(string[] args)
        {
            SQLiteConnection connection = new
SQLiteConnection("students.sqlite");//создаем подключение к БД

//создаем необходимые таблицы
            connection.CreateTable<Student>();
            connection.CreateTable<Group>();

            //добавим в таблицу groups новую группу
            connection.Insert(new Group() { Name = "Test" });

            //получаем rowId последней добавленной записи в
таблицу groups
            int lastId = connection.ExecuteScalar<int>("SELECT
last_insert_rowid()");

            //добавим в группу нового студента
            connection.Insert(new Student() { Name = "Иванов
Иван Иванович", GroupId = lastId });

            //считываем данные из таблиц
            var groups = connection.Table<Group>();
            foreach (Group db_group in groups)
            {
                Console.WriteLine($"{db_group.Id}            -
{db_group.Name}");
            }

            var students = connection.Table<Student>();
            foreach (Student db_student in students)
            {
                Console.WriteLine($"{db_student.Id}          -
{db_student.Name}");
            }
        }
    }
}
```

Задания и порядок выполнения работы

Задание 1. Выполнить все примеры, представленные в теоретическом материале.

Задание 2. Используя библиотеку SQLite.Net создать базу данных labs, которая содержит 1 таблицу labs с полями id, fio, nomer, labtheme, labstatus, где nomer – номер лабораторной, fio – ФИО, labtheme – содержит значением тему лабораторной работы, labstatus – отметку сдана работа или нет.

Задание 3. Внести в базу данных 3 произвольные записи с данными о своих лабораторных работах.

Контрольные вопросы

1. Что такое SQLite и каковы его основные особенности?
2. Как создать новую базу данных SQLite с помощью C#?
3. Как открыть существующую базу данных SQLite для чтения и записи с использованием C#?
4. Какие методы доступны для выполнения SQL-запросов к базе данных SQLite из кода C#?
5. Как добавить новую запись в таблицу SQLite с использованием C#?
6. Как удалить запись из таблицы SQLite с использованием C#?
7. Как обновить значение поля в записи таблицы SQLite с использованием C#?
8. Как получить все записи из таблицы SQLite в виде коллекции объектов C# с использованием LINQ?
9. Как обеспечить безопасность и целостность данных в SQLite при использовании в приложении C#?
10. Как выполнить транзакцию (группу операций) в SQLite с поддержкой атомарности, изоляции и надежности с использованием C#?

Лабораторная работа № 33

Тема: **«Основы работы с MySQL»**.

Цель: изучение принципов работы оператора цикла с параметром и циклов с условием в языке C#, а также отработка навыков их применения для написания программ.

Краткие теоретические сведения

1. Элемент управления Windows Forms Button

Элемент управления Windows Forms Button позволяет пользователю кликнуть его для выполнения действия.

1. Установка MySQLServer.

https://dev.mysql.com/downloads/windows/installer/8.0.html

– создание и настройка пользователей баз данных, конфигурация доступа

2. Установка MySQL Workbench (рис. 33.1, 33.2).

https://dev.mysql.com/downloads/workbench/

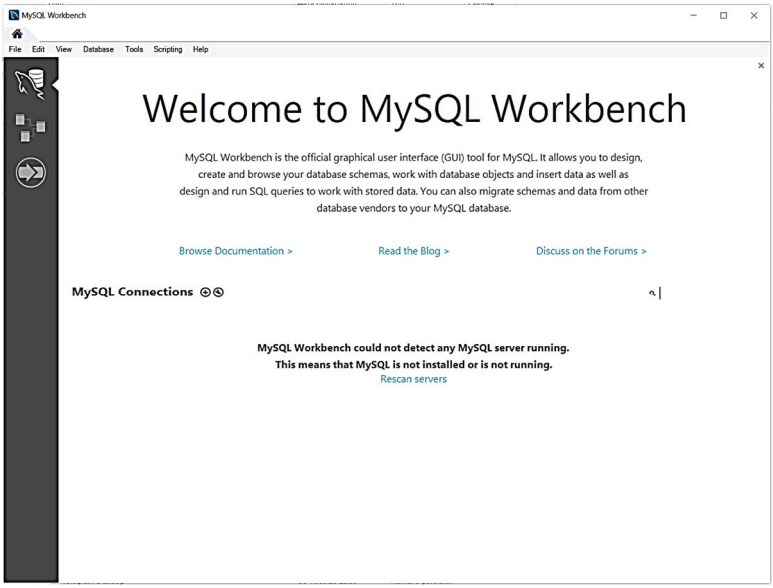

Рис. 33.1. Начальное окно MySQL Workbench

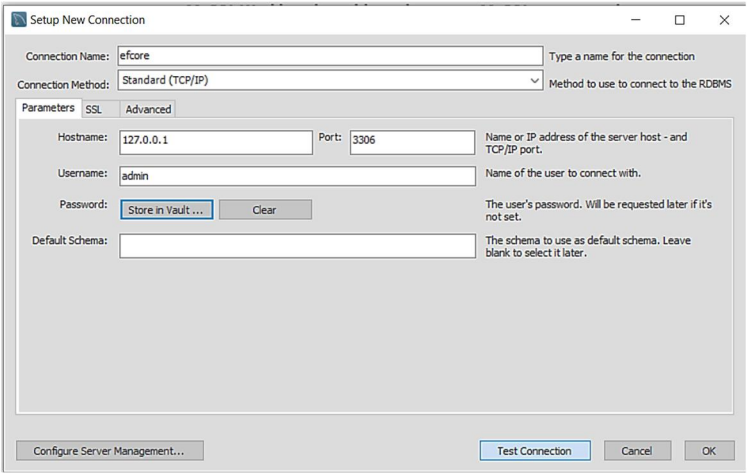

Рис. 33.2. Конфигурация подключения

3. Установка MySQLConnector
https://dev.mysql.com/downloads/windows/installer/8.0.html
или выбрать сразу все коннекторы при установке сервера
4. Установка пакета для MySQL через NuGet (рис. 33.3)
`Pomelo.EntityFrameworkCore.MySql`

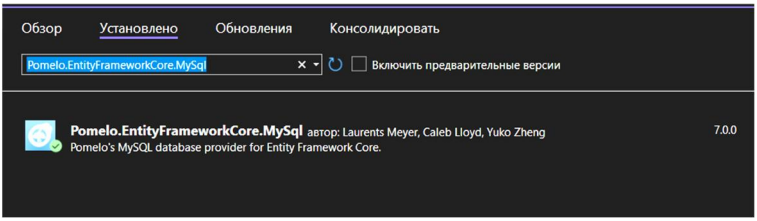

Рис.33.3. Установка пакетов

Далее, рассмотрим пример взаимодействия с базой данных

```
using Microsoft.EntityFrameworkCore;

namespace ConsoleApp18
{
    public class User
    {
        public int Id { get; set; }
        public string Name { get; set; }
        public int Age { get; set; }
    }
```

```
    public class ApplicationContext : DbContext
    {
        public DbSet<User> Users { get; set; }

        public ApplicationContext()
        {
            Database.EnsureCreated();
        }
        protected                override                void
OnConfiguring(DbContextOptionsBuilder optionsBuilder)
        {
            optionsBuilder.UseMySql(

"server=localhost;user=admin;password=test;database=users;",
                new MySqlServerVersion(new Version(8, 1, 0))
            );
        }
    }
    internal class Program
    {
        static void Main(string[] args)
        {
            using     (ApplicationContext     db     =     new
ApplicationContext())
            {
                User user1 = new User { Name = "Tom", Age = 33
};
                User user2 = new User { Name = "Alice", Age =
26 };

                db.Users.AddRange(user1, user2);
                db.SaveChanges();
            }
            // получение данных
            using     (ApplicationContext     db     =     new
ApplicationContext())
            {
                var users = db.Users.ToList();
                Console.WriteLine("Список объектов:");
                foreach (User u in users)
                {
                    Console.WriteLine($"{u.Id}.{u.Name}      -
{u.Age}");
                }
            }
        }
    }
}
```

В качестве строки инициализации использовали строку:

```
"server=localhost;user=admin;password=test;database=efcore;"
```

При создании подключения `efcore` использовали следующие параметры:

```
Connection name: efcore
Connection Method: Standard (TCP/IP)
Host Name: 127.0.0.1
Port: 3306
UserName: admin
Password: test
Default Schema: не задано (при необходимости, можем определить это свойство позднее)
```

Теперь необходимо кликнуть левой кнопкой мыши по созданному соединению (рис. 33.4).

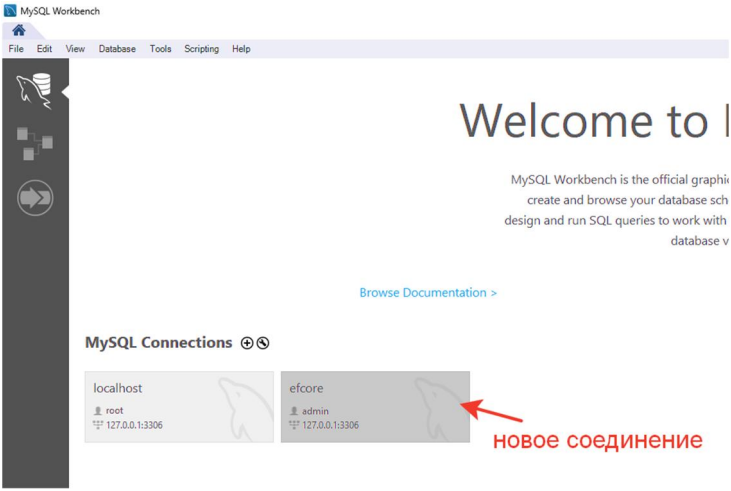

Рис. 33.4. Создание нового подключения

Подключившись к MySQL нам необходимо создать новую схему (базу данных MySQL) (рис. 33.5).

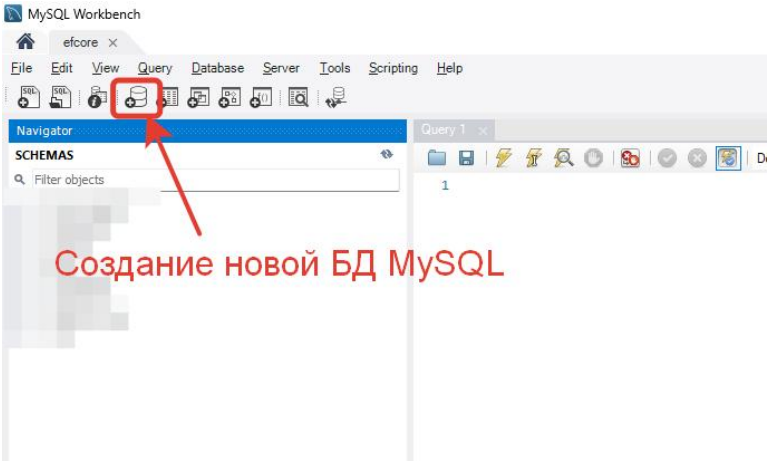

Рис. 33.5. Создание новой БД

Назовем нашу первую БД `users` и все остальные настройки новой схемы оставим по умолчанию (рис. 33.6):

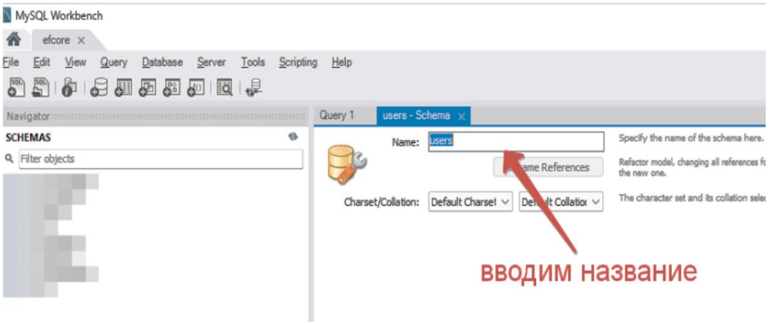

Рис. 33.6. Конфигурация БД

После того, как нажмем Apply, MySQL Workbench ещё раз в отдельном окне попросит подтвердить создание новой схемы – подтверждаем. После этого, в списке Schemas появится новая (пока ещё пустая) база данных (рис. 33.7):

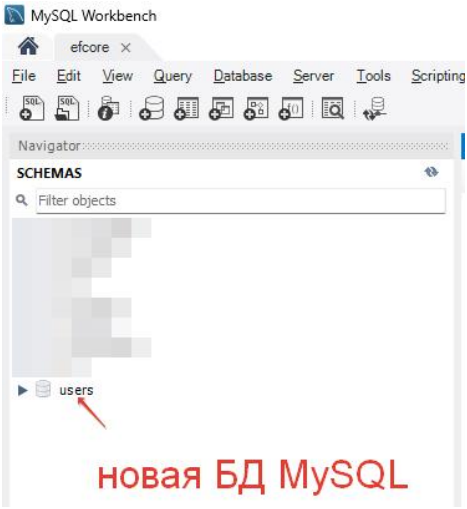

Рис. 33.7. Результат создания БД

На этом предварительная подготовка к созданию первого приложения с EF Core закончена. Теперь перейдем в Visual Studio и напишем наше приложение.

2. Создание консольного приложения с EF Core и MySQL

Итак, создадим новое консольное приложение в Visual Studio (рис. 33.8):

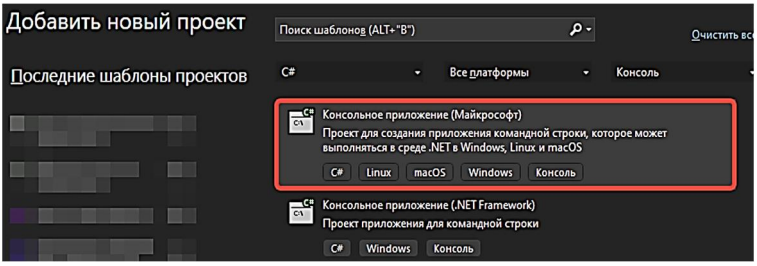

Рис. 33.8. Создание консольного приложения

Чтобы начать работать с EF Core и MySQL нам необходимо установить все необходимые пакеты. Для этого воспользуемся менеджером пакетов NuGet.

Первый пакет – `Microsoft.EntityFrameworkCore` (рис. 33.9):

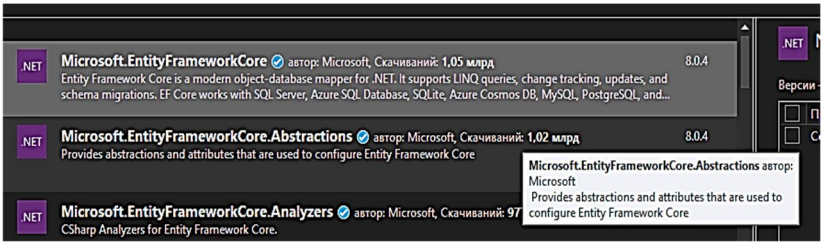

Рис. 33.9. Установка пакетов

Этот пакет добавит в наш проект основной функционал EF Core. Чтобы можно было работать в EF Core с MySQL, необходимо установить пакет, содержащий провайдер для этой базы данных. В перечне всех поставщиков баз данных на сайте Microsoft нам рекомендуется использовать провайдер, содержащийся в пакете `Pomelo.EntityFrameworkCore.MySql`. Устанавливаем этот пакет (рис. 33.10):

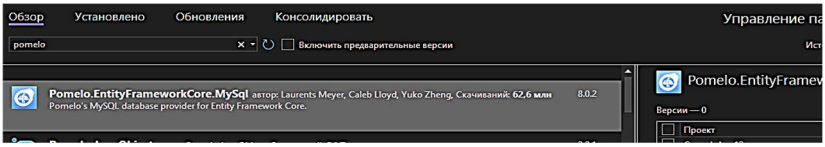

Рис. 33.10. Установка пакета `Pomelo.EntityFrameworkCore.MySql`.

Теперь все пакеты установлены и можно приступать к разработке логики нашего приложения.

3. Модель данных

EF Core использует модель метаданных для описания и сопоставления типов сущностей приложения с базой данных. По сути, моделью данных может выступать как один класс C#, так и группа классов, которые могут быть как связаны между собой, так и быть независимыми друг от друга. Так как вначале создали БД с именем users, то, соответственно, первым классом, описывающим модель данных, у нас будет класс User:

```
namespace EFCoreApp.Models
{
```

```
public class User
{
public int Id { get; set; }
public string? Name { get; set; }
public string? Email { get; set; }
public int Age { get; set; }
}
}
```

Как видим, наша модель – это обычный класс C#. Однако, так как объекты этого класса будут выступать сущностями EF Core, у класса определено свойство Id, которое будет выступать в качестве уникального ключа. Каждое свойство нашего класса будет в дальнейшем сопоставлено с отдельным столбцом таблицы БД. По умолчанию, при генерации таблиц БД EF Core в качестве первичных ключей рассматривает свойства с именами Id или с именами вида [Имя_класса]Id. То есть, в нашем случае, можно было бы определить свойство UserId и в EF Core это свойство использовалось бы по умолчанию как первичный ключ.

4. Создание контекста данных

Для взаимодействия с базой данных в EF Core используются специальные классы – контексты данных, которые наследуются от класса DbContext. Поэтому, создадим такой класс и назовем его ApplicationContext:

```
using Microsoft.EntityFrameworkCore;
using EFCoreApp.Models;
using Pomelo.EntityFrameworkCore.MySql;
namespace EFCoreApp
{
public class ApplicationContext: DbContext
{
private const string _connection =
@"server=127.0.0.1;Port=3306;user=admin;password=test;database
=users";
public DbSet<User> Users { get; set; }
public ApplicationContext()
{
_ = Database.EnsureCreated();
}
protected override void OnConfiguring(DbContextOptionsBuilder
optionsBuilder)
{
```

```
optionsBuilder.UseMySql(_connection,
ServerVersion.AutoDetect(_connection));
}
}
}
```

Рассмотрим по порядку, что содержит наш класс и как он работает. Во-первых, внутри класса определили константу, содержащую строку подключения к MySQL:

```
private const string _connection =
@"server=127.0.0.1;Port=3306;user=admin;password=test;database
=users";
```

Параметры подключения, которые использовали – представлены в разделе выше.

Далее, определили свойство:

```
public DbSet <User> Users { get; set; };
```

`DbSet` – это, как и `DbContext`, специальный класс, используемый в EF Core для хранения набора сущностей определенного класса. В нашем случае, модель состоит всего из одного класса, поэтому и свойство мы определили одно.

Далее, переопределили метод `OnConfiguring` класса `DbContext`. В этом методе настраиваем подключение к базе данных:

```
protected override void OnConfiguring(DbContextOptionsBuilder
optionsBuilder)
{
optionsBuilder.UseMySql(_connection,
ServerVersion.AutoDetect(_connection));
}
```

В этом методе используем параметр `optionsBuilder` класса `DbContextOptionsBuilder` для настройки подключения к MySQL. Метод `UseMySQL` имеет несколько переопределенных версий и, в нашем случае, принимает два параметра: первый – строка подключения и второй – версия сервера MySQL. Так как версия сервера MySQL может измениться или же мы вовсе можем не знать с какой версией сервера будем в дальнейшем работать, то для получения значения этого параметра мы использовали статический метод класса `ServerVersion.AutoDetect`, который и определит за нас версию сервера.

Так как на момент подключения к БД её в принципе может не существовать, то в конструкторе нашего класса используем

метод Database.EnsureCreated(), который проверяет наличие базы данных и, при необходимости, создает её:

```
public ApplicationContext()
{
_ = Database.EnsureCreated();
}
```

Создав контекст данных, можно переходить далее. Наполним нашу базу данных начальными данными

5. Присвоение начальных данных

Самый простой способ наполнить нашу базу данных начальными данными – это добавить эти данные до начала основной логики приложения. Например, это можно сделать следующим образом (в файле Program.cs нашего консольного приложения):

```
using EFCoreApp.Models;
namespace EFCoreApp
{
internal class Program
{
static void Main(string[] args)
{
using var db = new ApplicationContext();
if (db.Users.Any() == false) //нет никаких данных в БД
{
//создаем нового пользователя
User user = new()
{
Name = "admin",
Age = 30,
Email = "admin@corp.gov"
};
//добавляем запись в БД
db.Users.Add(user);
db.SaveChanges();
}
//получаем всех пользователей из БД
List<User> users = db.Users.ToList();
foreach (User user in users)
{
Console.WriteLine($"{user.Id}     {user.Name}     {user.Email}
{user.Age}");
}}}}
```

Опять же, рассмотрим по порядку, что написали. Первое:

```
using var db = new ApplicationContext();
```

Класс `ApplicationContext` через своего предка (базовый класс `DbContext`) реализует интерфейс `IDisposable`, поэтому для работы с `ApplicationContext` можно использовать конструкцию `using`. После того, как надобность в использовании класса отпадет, ресурсы, используемые классом, будут автоматически освобождены.

Далее, используя метод `LINQ` проверяем есть ли какая-либо информация в БД:

```
if (db.Users.Any() == false)
```

Если коллекция `Users` будет пуста, то метод `Any` вернет `False`. Соответственно, если коллекция пуста, то далее создаем объект класса `User`:

```
User user = new()
{
Name = "admin",
Age = 30,
Email = "admin@corp.gov"
};
```

И далее, добавляем новый объект в коллекцию и сохраняем изменения:

```
db.Users.Add(user);
db.SaveChanges();
```

После этого, получаем список пользователей из БД и выводим его в консоль:

```
List<User> users = db.Users.ToList();
foreach (User user in users)
{
Console.WriteLine($"{user.Id}     {user.Name}     {user.Email}
{user.Age}");
}
```

При первом запуске приложения увидим следующую информацию в консоли:

```
 1 admin admin@corp.gov 30
```

Стоит обратить внимание на то, что при создании объекта мы не присваивали значение свойству `Id`, однако, при записи объекта в БД это свойство было присвоено объекту автоматически.

Теперь можно снова вернуться в MySQL Workbench и посмотреть на содержимое нашей БД. Вот так выглядит теперь база данных (рис. 33.11):

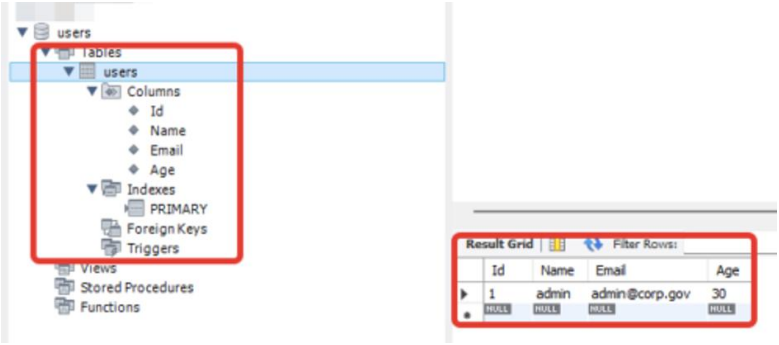

Рис. 33.11. Просмотр содержимого БД

Имя таблицы соответствует имени коллекции сущностей в контексте – в ApplicationContext коллекция имеет названия Users, а в БД создана таблица users. Имена полей таблицы соответствуют именам свойств класса. В DDL наша таблица будет выглядеть следующим образом (всё это сделано EF Core):

```
CREATE TABLE `users` (
`Id` int NOT NULL AUTO_INCREMENT,
`Name` longtext CHARACTER SET utf8mb4 COLLATE
utf8mb4_0900_ai_ci,
`Email` longtext CHARACTER SET utf8mb4 COLLATE
utf8mb4_0900_ai_ci,
`Age` int NOT NULL,
PRIMARY KEY (`Id`)
) ENGINE=InnoDB AUTO_INCREMENT=1 DEFAULT CHARSET=utf8mb4
COLLATE=utf8mb4_0900_ai_ci
```

Таким образом, Entity Framework Core обеспечивает простое управление сущностями из базы данных, избавляя нас от непосредственной работы с конкретной базой данных – создание БД, таблиц в ней и прочие рутинные операции выполняются EF Core на основании классов модели и контекста данных.

Задания и порядок выполнения работы

Задание 1. Выполнить все примеры по развертыванию и созданию приложения для работы с СУБД MySQL

Контрольные вопросы

1. Что такое MySQL, и в чем отличие между MySQL и другими системами управления базами данных?
2. Как подключиться к серверу MySQL с использованием C#?
3. Какие библиотеки и компоненты можно использовать для работы с MySQL в C#?
4. Как создать новую базу данных MySQL с использованием C#?
5. Как выполнить SQL-запрос к базе данных MySQL средствами C#?
6. Как вставить, обновить и удалить данные в таблице MySQL с помощью C#?
7. Как осуществить подготовку и выполнение параметризованных запросов в C# для безопасной работы с данными в MySQL?
8. Как обработать исключения и ошибки, возникающие при работе с MySQL в C#?
9. Как осуществить чтение данных из таблицы MySQL и преобразовать их в объекты C# (например, классы)?
10. Как реализовать асинхронный доступ к базе данных MySQL с использованием C#?

Лабораторная работа № 34

Тема: «Основные операции с данными в SQLite. Проектирование интерфейсов приложений».

Цель: изучить основные операции с данными в SQLite, а также особенности проектирования пользовательских интерфейсов для приложений, работающих с базами данных, на языке программирования C#.

Краткие теоретические сведения

CRUD – акроним, обозначающий четыре основные операции при работе с базами данных: создание (create), чтение (read), модификация (update), удаление (delete). Рассмотрим более детально CRUD в SQLite.NET.

Итак, пусть у нас есть класс Group, содержащий основную информацию о студенческой группе:

```
[Table("groups")]
public class Group
{
    [PrimaryKey, AutoIncrement]
    [Column("id")]
    public int Id { get; set; }
    [Column("name")]
    public string Name { get; set; }
}
```

Добавим в этот класс ещё одно свойство – комментарий, который можно будет оставлять для записи группы в БД:

```
[Table("groups")]
public class Group
{
    [PrimaryKey, AutoIncrement]
    [Column("id")]
    public int Id { get; set; }
    [Column("name")]
    public string Name { get; set; }
    [Column("comment")]
    public string Comment { get; set; }
```

Теперь рассмотрим основные операции при работе с БД SQLite. Начнем по порядку.

1. Create (создание новых записей в БД).

1.1. Метод Insert

Для создания новых записей в SQLite.NET используется метод Insert, имеющий несколько перегруженных версий. Самый простой вариант метода выглядит следующим образом:

```
public int Insert(object obj)
```

Метод возвращает количество строк, затронутых в результате выполнения запроса. В качестве параметра метод принимает объект, который необходимо добавить в базу данных. Например:

```
            SQLiteConnection connection = new
SQLiteConnection("students.sqlite");//создаем подключение к БД
            //создаем необходимые таблицы
            connection.CreateTable<Student>();
            connection.CreateTable<Group>();
            //добавим в таблицу groups новую группу
            connection.Insert(new Group()
            {
                Name = "ПБ-22",
                Comment ="Пожарная безопасность. Набор 2022
года"
            });
```

В данном случае, в базу будет добавлена новая запись:

```
SQLite новая запись
SQLite новая запись
```

Как можно видеть, выше была добавлена запись «Test». Это запись из прошлой части, поэтому поле comment у этой записи равно null.

Второй вариант этого метода позволяет добавлять в запрос INSERT дополнительные команды SQLite в соответствии с документацией. Например, можем выполнить такой метод:

```
Group group = new Group()
{
    Name = "ПБ-22",
    Comment = "Пожарная безопасность. Набор 2022 года"
};
connection.Insert(group, "OR FAIL");
```

В этом случае, запрос к БД будет составлен следующим образом:

```
INSERT OR FAIL INTO...
```

вместо обычного
`INSERT INTO....`

Команды `REPLACE`, `IGNORE`, `FAIL` и т.д., которые можно указать вместе с `INSERT` будут работать только при наличии в таблице уникального поля, не допускающего `null`. Также команды не сработают если в таблице определено всего одно уникальное поле с ключевым словом `AUTOINCREMENT`.

1.2. Метод InsertOrReplace

Этот метод выполняет запрос к БД типа «`INSERT OR REPLACE INTO....`». При вставке нового элемента в таблицу могут нарушаться ограничения уникальности (`UNIQUE`) или первичного ключа (`PRIMARY KEY`) и, в этом случае можем получить исключение в своем приложении. Метод `InsertOrReplace` позволяет избежать этого момента следующим образом: если в таблице встречается запись, которая может вызвать конфликт, то эта запись удаляется, а на её место вставляется новая.

Чтобы продемонстрировать как работает этот метод, сделаем уникальным поле `name` в таблице `groups`. Для начала, допишем класс `Group` следующим образом:

```
[Table("groups")]
public class Group
{
    [PrimaryKey, AutoIncrement]
    [Column("id")]
    public int Id { get; set; }
    [Column("name"), Unique]
    public string Name { get; set; }
    [Column("comment")]
    public string Comment { get; set; }
}
```

Добавили к свойству класса `Name` атрибут `Unique`, который указывает, что поле в таблице должно быть уникальным. Теперь, если у вас рядом с `exe`-файлом уже есть файл `students.sqlite`, то его необходимо удалить.

Теперь попробуем добавить две одинаковые записи в таблицу `groups`:

```
SQLiteConnection connection = new
SQLiteConnection("students.sqlite");//создаем подключение к БД
```

```
 //создаем необходимые таблицы
connection.CreateTable<Student>();
connection.CreateTable<Group>();
//создаем объект
Group group = new Group()
{
    Name = "ПБ-22",
    Comment = "Программная инженерия."
};
//пробуем добавить объект дважды в таблицу
connection.Insert(group);
connection.InsertOrReplace(group);
```

Так как мы пытаемся добавить в таблицу две одинаковые записи, то первая запись добавиться в таблицу, а, затем будет удалена и на её место будет вставлена вторая запись.

Если вам необходимо, чтобы запись с нарушением уникальности не заменяла старую, а игнорировалась при добавлении в таблицу, то воспользуйтесь методом `Insert` с дополнительной командой `OR IGNORE`.

1.3. Метод InsertAll

Этот метод, как и метод `Insert` добавляет новые записи в таблицу БД, но, при этом, принимает в качестве первого параметра не один, а набор элементов (любой тип, реализующий интерфейс `IEnumerable`). Например, можно добавить сразу две группы в таблицу `groups`:

```
List<Group> groups = new List<Group>()
{
new Group()
{
    Name = "ПБ-222",
    Comment = "Пожарная безопаность. Набор 2022 года"
},
new Group()
{
    Name = "ТБ-222",
    Comment = "Техносферная безопаность. Набор 2022 года"
}
};
            //добавляем два элемента в таблицу
connection.InsertAll(groups);
```

Следует отметить, что при использовании метода, как показано выше, запросы на добавление набора объектов будут проводиться в рамках одной транзакции, что, в принципе, практически избавляет нас от лишних операций по созданию/завершению транзакций и повышения производительности работы SQLite. Однако, если вам необходимо, чтобы транзакция не создавалась, то метод можно вызвать с двумя параметрами:

```
connection.InsertAll(groups, false);
```

Второй параметр указывает на то, что записи должны добавляться в таблицу без создания транзакции.

2. Read (чтение записей). Метод Table<T>

Метод Table возвращает объект типа TableQuery<T>, который имеет следующее описание:

```
public class TableQuery<T> : BaseTableQuery, IEnumerable<T>,
IEnumerable
```

В классе определены необходимые методы для манипуляции полученной выборкой данных. Например:

```
var data = connection.Table<Group>().Where(f => f.Name == "ПБ-
222");
foreach (var group in data)
{
    Console.WriteLine(group.Comment);
}
```

Так выбрали из таблицы запись, у которой поле name равно значению «ПБ-222». Используя методы класса TableQuery, а также методы LINQ, можно фильтровать данные, сортировать, удалять, представлять в виде массива или списка и т.д.

3. Update (обновление записей). Методы Update и UpdateAll

Оба метода обновляют записи в БД. Соответственно, обязательным условием работы этих методов является, то, что у объектов, которые необходимо обновить обязательно должен быть определен первичный ключ (свойство с атрибутом PrimaryKey). Например, следующий вызов метода Update не приведет к обновлению записи:

```
Group group = new Group()
{
    Name = "ТБ-222",
    Comment = "Новый комментарий"
};
connection.Update(group);
```

Так как мы не определили значение свойства Id, которое является первичным ключом. Правильный вызов метода:

```
Group group = new Group()
{
    Id = 2, //указываем Id записи, которую необходимо обновить
    Name = "ТБ-222",
    Comment = "Новый коммент. Набор 2020 года"
};
connection.Update(group);
```

В отличие от метода Insert, метод Update не имеет перегруженной версии для добавления дополнительных команд в запрос типа UPDATE OR IGNORE, хотя, судя по документации SQLite могли бы это сделать.

4. Delete (удаление записей)

4.1. Метод SqliteConnection.Delete

С помощью этого метода можно удалить отдельную запись из таблицы. При этом, у удаляемого объекта должно быть определено свойство, для которого установлен атрибут PrimaryKey. Например:

```
Group group = new Group()
{
    Id = 2, //указываем Id записи, которую необходимо удалить
};
connection.Delete(group);
//ИЛИ
connection.Delete<Group>(2);
```

Так как мы удаляем запись, то при создании объекта Group нам достаточно указать только значение первичного ключа. Во втором случае, воспользовались универсальным методом Delete в параметрах которого передали значение первичного ключа. В обоих вариантах результат будет одинаковый – удалиться запись с id=2.

4.2. Метод TableQuery.Delete

Этот метод удаляет все записи из таблицы, которые были затронуты текущим запросом. Например, если мы хотим удалить все записи из таблицы, то нам достаточно вызвать последовательно два метода – Table и Delete следующим образом:

```
connection.Table<Group>().Delete();
```

Метод Table вернет нам объект типа TableQuery, а далее уже вызываем метод Delete полученного объекта. При этом, можем воспользоваться перегруженной версией этого метода и удалить только те записи, которые соответствуют заданному условию, например:

```
connection.Table<Group>().Delete(f=>f.Id>4);
```

В этом случае вначале, используя метод Table, выбираем записи из таблицы groups, а затем – удаляем все записи из полученной выборки, у которых поле id больше 4.

Задания и порядок выполнения работы

Задание 1. Разработать графический интерфейс для выполнения основных операций с данными в базе данных, разработанной в лабораторной работе № 33.

Задание 2. Реализовать форму поиска по базе данных с возможностью фильтрации по различным полям.

Задание 3. Создать интерфейс для добавления новых записей в таблицу с проверкой правильности ввода данных.

Задание 4. Разработать форму для удаления записей из таблицы с подтверждением операции.

Задание 5. Реализовать интерфейс для редактирования структуры таблицы (добавление и удаление полей).

Контрольные вопросы

1. Что такое SQLite и каковы его основные преимущества?
2. Какие типы данных поддерживает SQLite?
3. Опишите процесс подключения к базе данных SQLite из приложения на C#?
4. Как создать новую таблицу в SQLite с использованием C#?
5. Как вставить данные в таблицу SQLite с помощью C#?

6. Как выполнить запрос к SQLite базе данных с использованием C#?

7. Опишите процесс обновления и удаления данных из таблицы SQLite с использованием C#.

8. Как получить результаты запроса из SQLite базы данных в виде объекта DataTable с использованием C#?

9. Как обеспечить безопасность и целостность данных при работе с SQLite в C# приложении?

10. Как правильно закрыть соединение с базой данных SQLite после завершения работы с ней в C# коде?

Лабораторная работа № 35

Тема: **«Асинхронное программирование»**

Цель: изучить принципы работы асинхронных программ в C#.

Краткие теоретические сведения

1. Асинхронное программирование в C#

Асинхронное программирование позволяет избежать появления узких мест в приложении и увеличить общую скорость реагирования на действия пользователя. Суть асинхронного программирования заключается в том, что отдельные операции в вашем коде выносятся в специальные асинхронные методы и выполняются отдельно таким образом, чтобы ресурсы вашего приложения использовались максимально эффективно.

1.1. Синхронное выполнение задач

Чтобы продемонстрировать преимущества использования приемов асинхронного программирования в C# рассмотрим бытовой пример, с которым Вы могли встречаться ни один раз, будучи студентами ВУЗов – работу частной мелкой типографии.

Ситуация: в офисе один работник (Вы). У вас есть в распоряжении плоттер, черно-белый и цветной принтер, ну и по мелочи всякие типографские штуки. На дворе, допустим, конец мая (время сессии). Приходит к вам студент с целой стопкой распечатанных листов формата A4 и флэшкой и выдает вам такой заказ:

На флэшке три файла – (1) чертеж на формате A1. Его надо распечатать на плоттере; (2) Документ Word – его надо распечатать в черно-белом цвете на A4; (3) Документ PDF с массой картинок – его надо распечатать в цвете на формате A3. Распечатанные листы сложить вместе с теми, которые принес студент. Сброшюровать все листы A4.

Итого, получаем пять задач. Как будет действовать в этом случае «однозадачный» человек? Он начнет выполнять задачи синхронно – сначала распечатает одно, потом другое, третье, потом сложит все листы A4 в стопку, сброшюрует их и, наконец-

то, отдаст заказ. Посмотрим, как эти задачи мы бы смоделировали в C#:

```
using System;
using System.Threading.Tasks;
using System.Diagnostics;
namespace Typography
{
    //Классы для примера. Они не несут в себе никакой ценной
идеи и сделаны просто для демонстрации процесса
    internal class ListsA1 { } //распечатаные листы A1
    internal class ListsA3 { } //распечатаные листы A3
    internal class ListsA4 { } //распечатаные листы A4
    internal class ListsHeap { } //собранная стопка бумаги
    internal class Booklet { } //готовая брошюра формата A4
    class Program
    {
        //методы (рабочие операции работника типографии)
        //Распечатка листов формата A1
        private static ListsA1 PrintListA1(int count)
        {
            Console.WriteLine("Включаем плоттер");
            Console.WriteLine("Ждем несколько секунд...");
            Task.Delay(3000).Wait();
            Console.WriteLine("Печатаем формат A1 на
плоттере");
            for (int i = 0; i < count; i++)
            {
                Console.WriteLine($"Распечатали {i + 1} лист
A1");
                Task.Delay(1000);
            }
            Console.WriteLine("Аккуратно сворачиваем листы
A1");
            Task.Delay(3000).Wait();
            Console.WriteLine("Распечатка листов формата A1
закончена");
            return new ListsA1();
        }
        private static ListsA3 PrintListA3(int count)
        {
            Console.WriteLine("Включаем цветной принтер");
            Console.WriteLine("Ждем несколько секунд...");
            Task.Delay(3000).Wait();
            Console.WriteLine("Печатаем формат A3 на цветном
принтере");
            for (int i = 0; i < count; i++)
```

```
            {
                Console.WriteLine($"Распечатали {i + 1} лист
A3");
                Task.Delay(500);
            }
            Console.WriteLine("Аккуратно сворачиваем листы
A3");
            Task.Delay(3000).Wait();
            Console.WriteLine("Распечатка листов формата A3
закончена");
            return new ListsA3();
        }
        private static ListsA4 PrintListA4(int count)
        {
            Console.WriteLine("Включаем черно-белый принтер");
            Console.WriteLine("Ждем несколько секунд...");
            Task.Delay(3000).Wait();
            Console.WriteLine("Печатаем формат A4 на черно-
белом принтере");
            for (int i = 0; i < count; i++)
            {
                Console.WriteLine($"Распечатали {i + 1} лист
A4");
                Task.Delay(100);
            }
            Console.WriteLine("Аккуратно складываем листы в
стопку");
            Task.Delay(3000).Wait();
            Console.WriteLine("Распечатка листов формата A4
закончена");
            return new ListsA4();
        }
        private static ListsHeap CreateHeap(ListsA4 lists)
        {
            Console.WriteLine("Берем все листы A4");
            Console.WriteLine("Аккуратно складываем
вместе...");
            Task.Delay(5000).Wait();
            Console.WriteLine("Стопка бумаги готова. Можно
брошюровать");
            return new ListsHeap();
        }
        private static Booklet CreateBooklet(ListsHeap heap)
        {
            Console.WriteLine("Берем стопку листов A4");
            Console.WriteLine("Брошюруем...");
            Task.Delay(5000).Wait();
```

```
            Console.WriteLine("Брошюра готова");
            return new Booklet();
        }
        static void Main(string[] args)
        {
            Stopwatch stopwatch = Stopwatch.StartNew();
            ListsA1 listsA1 = PrintListA1(2); //печатаем два
листа A1
            Console.WriteLine("------Задача по распечатке
листов A1 выполнена------");
            ListsA3 listsA3 = PrintListA3(4); //печатаем
четыре листа A3
            Console.WriteLine("------Задача по распечатке
листов A3 выполнена------");
            ListsA4 listsA4 = PrintListA4(10);//печатаем 10
листов A4
            Console.WriteLine("------Задача по распечатке
листов A4 выполнена------");
            ListsHeap heap = CreateHeap(listsA4); //складываем
все листы вместе
            Console.WriteLine("------Стопка листов A4 готова--
----");
            Booklet booklet = CreateBooklet(heap);//собираем
брошюру
            Console.WriteLine("------Листы A4 сброшюрованы----
--");
            Console.WriteLine("------Поздравляем с успешно
выполненным первым заказом!------");
            stopwatch.Stop();
            Console.WriteLine($"Заказ выполнен за
{stopwatch.Elapsed.TotalSeconds} секунд");
        }
    }
}
```

Каждый метод, относящийся к распечатке документа – `PrintListA1`, `PrintListA3`, `PrintListA4` эмулирует работу с оборудованием –включаем принтер/плоттер, ждем пока он проведет самодиагностику, потом печатаем. Каждый распечатанный лист тратит наше драгоценное время по-разному – дольше всех печатается формат A1, быстрее всех – A4. Дальше идут два метода, которые используют результаты предыдущих работ: в метод `CreateHeap` мы должны передать все распечатанные листы A4, чтобы собрать стопку бумаги, а в метод `CreateBooklet` –передаем аккуратно сложенную стопку бумаги, чтобы собрать из

этой стопки буклет. Все эти операции выполняем последовательно в методе `Main()`. Результат работы такой программы будет следующим:

```
Включаем плоттер
Ждем несколько секунд…
Печатаем формат A1 на плоттере
Распечатали 1 лист A1
Распечатали 2 лист A1
Аккуратно сворачиваем листы A1
Распечатка листов формата A1 закончена
——Задача по распечатке листов A1 выполнена——
Включаем цветной принтер
Ждем несколько секунд…
Печатаем формат A3 на цветном принтере
Распечатали 1 лист A3
……….
Распечатали 4 лист A3
Аккуратно сворачиваем листы A3
Распечатка листов формата A3 закончена
——Задача по распечатке листов A3 выполнена——
Включаем черно-белый принтер
Ждем несколько секунд…
Печатаем формат A4 на черно-белом принтере
Распечатали 1 лист A4
Распечатали 2 лист A4
…….
Распечатали 9 лист A4
Распечатали 10 лист A4
Аккуратно складываем листы в стопку
Распечатка листов формата A4 закончена
——Задача по распечатке листов A4 выполнена——
Берем все листы A4
Аккуратно складываем вместе…
Стопка бумаги готова. Можно брошюровать
——Стопка листов A4 готова——
Берем стопку листов A4
Брошюруем…
Брошюра готова
——Листы A4 сброшюрованы——
——Поздравляем с успешно выполненным первым заказом!——
Заказ выполнен за 28,0748336 секунд
```

Итого, мы затратили на всё про всё почти 30 секунд виртуального времени. Каждая задача выполняется синхронно и, пока очередная задача не будет выполнена, мы не можем перейти к следующей и, в итоге, время задач у нас суммируется.

1.2. Асинхронное выполнение задач

В таком виде, как представлено выше, код блокирует выполняющий (главный) поток, не позволяя выполнять другие действия. Выполнение такого кода не будет прервано до тех пор, пока все операции не будут выполнены. Это, согласитесь, не самый оптимальный вариант написания приложений – фактически, наш процессор простаивает впустую, не выполняя никакой работы, пока мы «печатаем». Это все равно, что стоять и смотреть на принтер пока из него лезут распечатанные листы и, при этом, игнорировать всё и всех вокруг. Попробуем изменить наш код так, чтобы он выполнялся асинхронно. Ключевое слово await позволяет обойтись без блокировки главного потока для запуска задачи, а затем продолжить выполнение, когда задача завершается.

Перепишем пример. Ниже представлены только измененные участки кода:

```
using System.Diagnostics;

private static async Task<ListsA1> PrintListA1Async(int count)
{
    Console.WriteLine("Включаем плоттер");
    Console.WriteLine("Ждем несколько секунд...");
    await Task.Delay(3000);
    Console.WriteLine("Печатаем формат A1 на плоттере");
    for (int i = 0; i < count; i++)
    {
        Console.WriteLine($"Распечатали {i + 1} лист A1");
        await Task.Delay(1000);
    }
    Console.WriteLine("Аккуратно сворачиваем листы A1");
    await Task.Delay(3000);
    Console.WriteLine("Распечатка листов формата A1
закончена");
    return new ListsA1();
}
private static async Task<ListsA3> PrintListA3Async(int count)
{
    Console.WriteLine("Включаем цветной принтер");
    Console.WriteLine("Ждем несколько секунд...");
    await Task.Delay(3000);
    Console.WriteLine("Печатаем формат A3 на цветном
принтере");
    for (int i = 0; i < count; i++)
    {
```

```
        Console.WriteLine($"Распечатали {i + 1} лист А3");
        await Task.Delay(500);
    }
    Console.WriteLine("Аккуратно сворачиваем листы А3");
    await Task.Delay(3000);
    Console.WriteLine("Распечатка листов формата А3
закончена");
    return new ListsA3();
}
private static async Task<ListsA4> PrintListA4Async(int count)
{
    Console.WriteLine("Включаем черно-белый принтер");
    Console.WriteLine("Ждем несколько секунд...");
    await Task.Delay(3000);
    Console.WriteLine("Печатаем формат А4 на черно-белом
принтере");
    for (int i = 0; i < count; i++)
    {
        Console.WriteLine($"Распечатали {i + 1} лист А4");
        await Task.Delay(100);
    }
    Console.WriteLine("Аккуратно складываем листы в стопку");
    await Task.Delay(3000);
    Console.WriteLine("Распечатка листов формата А4
закончена");
    return new ListsA4();
}
Метод Main
static async Task Main(string[] args)
{
    Stopwatch stopwatch = Stopwatch.StartNew();
    ListsA1 listsA1 = await PrintListA1Async(2); //печатаем два
листа А1
    Console.WriteLine("------Задача по распечатке листов А1
выполнена------");
    ListsA3 listsA3 = await PrintListA3Async(4); //печатаем
четыре листа А3
    Console.WriteLine("------Задача по распечатке листов А3
выполнена------");
    ListsA4 listsA4 = await PrintListA4Async(10);//печатаем 10
листов А4
    Console.WriteLine("------Задача по распечатке листов А4
выполнена------");
    ListsHeap heap = CreateHeap(listsA4); //складываем все
листы вместе
    Console.WriteLine("------Стопка листов А4 готова------");
    Booklet booklet = CreateBooklet(heap);//собираем брошюру
```

```
    Console.WriteLine("------Листы А4 сброшюрованы------");
    Console.WriteLine("------Поздравляем с успешно выполненным
первым заказом!------");
    stopwatch.Stop();
    Console.WriteLine($"Заказ выполнен за
{stopwatch.Elapsed.TotalSeconds} секунд");
}
```

Во-первых, у методов `Print...` появилось ключевое слово `async`. Это ключевое слово означает, что внутри метода могут встречаться асинхронные операции. То есть, наличие `async` в определении метода совсем не гарантирует, что он автоматически станет выполняться асинхронно, асинхронным метод можно будет назвать только в том случае, если в теле метода встретится хотя бы один раз другое ключевое слово – `await`. Оператор `await` в методе говорит о том, что операцию будет выполняться асинхронно и выполнение этой операции не будет блокировать главный поток. В нашем случае, асинхронно будут выполняться задержки времени (`Task.Delay()`)

Во-вторых, в названии методов добавился суффикс `Await`, например, `PrintListA3Async`. Так можно сообщить другому программисту, что перед нами асинхронный метод.

Отдельно стоит отметить, что `async/await` используются в паре. Причем, если вы используете только `async`, то компилятор `C#` вас просто предупредит, что метод все равно будет выполняться синхронно, так как не обнаружено ни одного `await`. А вот `await` вы уже отдельно от `async` использовать не сможете вообще – это приведет к ошибке вида:

```
Ошибка CS4032 Оператор await можно использовать только в методах
с модификатором async. Consider marking this method with the
'async' modifier and changing its return type to 'Task'.
```

Ну и, в-третьих, обратите внимание на метод `Main` – он теперь тоже стал асинхронным, так как внутри него мы планируем вызывать асинхронные методы. Проверим, как сработает теперь наша типография:

```
Включаем плоттер
Ждем несколько секунд…
Печатаем формат А1 на плоттере
Распечатали 1 лист А1
Распечатали 2 лист А1
Аккуратно сворачиваем листы А1
```

```
Распечатка листов формата A1 закончена
--Задача по распечатке листов A1 выполнена—
Включаем цветной принтер
Ждем несколько секунд…
Печатаем формат A3 на цветном принтере
Распечатали 1 лист A3
……
Распечатали 4 лист A3
Аккуратно сворачиваем листы A3
Распечатка листов формата A3 закончена
—-Задача по распечатке листов A3 выполнена—
Включаем черно-белый принтер
Ждем несколько секунд…
Печатаем формат A4 на черно-белом принтере
Распечатали 1 лист A4
Распечатали 2 лист A4
Распечатали 3 лист A4
…….
Распечатали 10 лист A4
Аккуратно складываем листы в стопку
Распечатка листов формата A4 закончена
—-Задача по распечатке листов A4 выполнена—
Берем все листы A4
Аккуратно складываем вместе…
Стопка бумаги готова. Можно брошюровать
—-Стопка листов A4 готова—
Берем стопку листов A4
Брошюруем…
Брошюра готова
—-Листы A4 сброшюрованы—
—-Поздравляем с успешно выполненным первым заказом!—
Заказ выполнен за 33,2608234 секунд
```

Обращаем внимание на результат – время выполнения операции не только не сократилось, но еще и стало больше!

Возникает логичный вопрос: зачем тогда мы использовали async/await? Так, при выполнении задач главный поток приложения не блокировался. Задачи выполнялись отдельно и, поэтому, была возможность выполнять другие задачи. Конечно, в консольном приложении этот эффект от использования async/await практически не заметен, но, если переписать приложение, используя графический интерфейс можно убедиться, что без асинхронности приложение «зависнет». Увеличение времени связано всё же с особенностями использования Task.

Для некоторых приложений такого подхода бывает достаточно, чтобы «оживить» приложение. Возвращаясь к аналогии с типографией, все еще остается необходимость ждать пока очередной принтер выполнит свою работу, но приложение уже может взаимодействовать, например, отправляя сообщение, что скоро всё будет готово.

1.3. Одновременный запуск асинхронных задач

Опять же, по аналогии с нашим примером: в реальной ситуации мы бы вряд ли ждали пока все принтеры распечатают все листы и только потом бы начали собирать брошюру. Мы бы сделали так: запустили бы сразу все печатные устройства, затем дождались бы пока отработает черно-белый принтер и, не дожидаясь плоттера и цветного принтера начали бы собирать брошюру. Так мы выполнили бы заказ быстрее. Давайте так и сделаем в нашем примере. Перепишем метод `Main` следующим образом:

```
using System.Diagnostics;

static async Task Main(string[] args)
{
    Stopwatch stopwatch = Stopwatch.StartNew();
    Task<ListsA1>    listA1_Task    =    PrintListA1Async(2);
//запускаем принтер в работу (создаем задачу)
    Task<ListsA3>                listsA3_Task                =
PrintListA3Async(4);//запускаем  принтер  в  работу  (создаем
задачу)
    Task<ListsA4>                listsA4_Task                =
PrintListA4Async(10);//запускаем  принтер  в  работу  (создаем
задачу)
                                                          //листы
A4 нам нужны для стопки и брошюрования
    ListsA4 listsA4 = await listsA4_Task; //ждем пока отработает
черно-белый принтер
    Console.WriteLine("------Задача  по  распечатке  листов  A4
выполнена------");
    ListsHeap  heap  =  CreateHeap(listsA4);  //складываем  все
листы вместе
    Console.WriteLine("------Стопка листов A4 готова------");
    Booklet booklet = CreateBooklet(heap);//собираем брошюру
    Console.WriteLine("------Листы A4 сброшюрованы------");
```

```
    //эти задачи пусть выполняются отдельно - они нам не мешают
возиться с листами A4
    ListsA1 listsA1 = await listA1_Task;
    Console.WriteLine("------Задача по распечатке листов A1
выполнена------");
    ListsA3 listsA3 = await listsA3_Task;
    Console.WriteLine("------Задача по распечатке листов A3
выполнена------");
    Console.WriteLine("------Поздравляем с успешно выполненным
первым заказом!------");
    stopwatch.Stop();
    Console.WriteLine($"Заказ выполнен за
{stopwatch.Elapsed.TotalSeconds} секунд");
}
}
```

Собственно, в комментариях к коду все сказано –создали несколько задач, затем одну задачу подождали, так как от её результата зависит выполнение двух синхронных операций, а выполнение остальных задач, так как они оказались достаточно продолжительными, вынесли в конец метода – пусть работают и не мешают. Вот какой результат в итоге получим:

```
Включаем плоттер
Ждем несколько секунд…
Включаем цветной принтер
Ждем несколько секунд…
Включаем черно-белый принтер
Ждем несколько секунд…
Печатаем формат A1 на плоттере
Распечатали 1 лист A1
Печатаем формат A3 на цветном принтере
Распечатали 1 лист A3
Печатаем формат A4 на черно-белом принтере
Распечатали 1 лист A4
Распечатали 2 лист A4
Распечатали 3 лист A4
Распечатали 4 лист A4
Распечатали 5 лист A4
Распечатали 2 лист A3
Распечатали 6 лист A4
Распечатали 7 лист A4
Распечатали 8 лист A4
Распечатали 9 лист A4
Распечатали 10 лист A4
Распечатали 2 лист A1
Распечатали 3 лист A3
```

```
Аккуратно складываем листы в стопку
Распечатали 4 лист A3
Аккуратно сворачиваем листы A1
Аккуратно сворачиваем листы A3
Распечатка листов формата A4 закончена
——Задача по распечатке листов A4 выполнена——
Берем все листы A4
Аккуратно складываем вместе…
Распечатка листов формата A1 закончена
Распечатка листов формата A3 закончена
Стопка бумаги готова. Можно брошюровать
——Стопка листов A4 готова——
Берем стопку листов A4
Брошюруем…
Брошюра готова
——Листы A4 сброшюрованы——
——Задача по распечатке листов A1 выполнена——
——Задача по распечатке листов A3 выполнена——
——Поздравляем с успешно выполненным первым заказом!——
Заказ выполнен за 17,1642155 секунд
```

Обратите внимание на то, что принтеры работали одновременно и не мешали друг другу, а вот собирать брошюру мы начали только после того, как были распечатаны все листы A4. Результат – время выполнения заказа сократилось с 28 секунд в синхронном режиме до 17 секунд в асинхронном. Здесь мы запустили асинхронные задачи, но не стали ожидать их результата, а сразу перешли далее. В итоге, выполнение кода у нас остановилось на строке:

```
ListsA4 listsA4 = await listsA4_Task; //ждем пока отработает
черно-белый принтер
```

То есть, задачи `listA1_Task`, `listA3_Task` и `listA4_Task` продолжили выполняться в отдельных потоках, но мы указали оператором `await`, что необходимо дождаться результата одной из задач. И мы его дождались. Далее выполнились два синхронных метода и, пока выполняли эти методы, у нас отработали задачи `listA1_Task` и `listA3_Task` (обратите внимание на лог программы). Когда собрали брошюру (то есть отработал синхронный метод `CreateBooklet`), вернулись к принтеру и плоттеру и забрали распечатки – снова выполнили `await`, но так как на этот момент времени задачи уже выполнили свою работу, то результат был получен практически мгновенно.

Вот таким способом мы добились асинхронности выполнения кода и ускорили его выполнение.

2. Преимущества и недостатки асинхронного программирования

Какие можно сделать предварительные выводы по поводу асинхронного программирования.

ЗА асинхронность	ПРОТИВ асинхронности
Асинхронность позволяет ускорить обработку длительных операций, запустив их выполнение одновременно	Использование `async/await` имеет свои накладные расходы, поэтому использовать асинхронность стоит там, где это действительно необходимо (в операциях ввода-вывода, при работе с большими файлами и т.д.). По незнанию можно легко «затормозить» время выполнения работы программой, хотя интерфейс «виснуть» не будет
В отличие от чистой многопоточности асинхронные методы писать, проще и быстрее	Читать код с `async/await` сложнее, чем обычный синхронный код

Таким образом, для того, чтобы метод был асинхронным он должен: а) содержать ключевое слово `async` в определении; б) в теле метода должна быть операция с оператором `await`, который приостанавливает вычисление асинхронного метода до завершения асинхронной операции. Наличие только `async` не делает метод асинхронным.

3. Результаты, возвращаемые асинхронным методом

Рассмотрим какие результаты возвращает асинхронный метод и как с этими результатами можно работать.

Согласно документации Microsoft, асинхронный метод может возвращать значения следующих типов:

```
void
Task
Task<TResult>
```

Для работы, обычно бывает достаточным использовать первые три типа данных их и рассмотрим. Итак, в каких случаях

какой тип возвращаемых асинхронным методом результатов следует предпочитать и использовать.

3.1. Void

В этом случае асинхронный метод ничего не возвращает. Например, можно создать такой асинхронный метод:

```
private static async void DoWork()
{
    await Task.Delay(3000);
    Console.WriteLine("Работа выполнена");
}
```

В этом случае метод ничего не возвращает. Однако не стоит злоупотреблять void при написании асинхронных методов, даже в том случае, если вы (на первый взгляд) ничего не ожидаете получить в результатах. Вот какие недостатки скрываются за использованием методов async void:

– исключения, вызываемые в методе async void, невозможно перехватывать вне этого метода.

– методы async void очень трудно тестировать.

– методы async void могут быть потенциально опасными, если вызывающий объект не ожидает, что они будут асинхронными.

Ну и ещё один момент: к методу async void нельзя применить оператор await.

Этот тип данных стоит применять в асинхронных методах только в одном случае – если вы пишете асинхронный обработчик события. В этом случае, использование в качестве возвращаемого результата void – единственный способ создать обработчик. Во всех остальных случаях следует использовать или Task или Task<TResult>.

3.2. Task

Этот тип используется в том случае, если в методе отсутствуют инструкции return или же имеются инструкции return, которые ничего не возвращают (прерывают выполнение метода). Дело в том, что даже, если ваш асинхронный метод не возвращает какого-либо результата – это совсем не означает, что

результат выполнения асинхронной задачи отсутствует. Рассмотрим предыдущий пример, но перепишем его следующим образом:

```
private static async Task DoWork()
{
await Task.Delay(3000);
Console.WriteLine("Работа выполнена");
}
```

Теперь вызовем метод в нашем приложении:

```
static async Task Main(string[] args)
{
Task task = DoWork();
await task;
if (task.IsCompleted)
{
Console.WriteLine("Задача выполнена успешно");
}
else
{
Console.WriteLine("Что-то случилось...");
}
}
```

Как видите, в нашем методе нет `return`, то есть метод, по идее, ничего не должен бы возвращать, но он все равно возвращает объект типа `Task`, в котором содержится полезная информация о задаче. И в данном примере проверяем выполнена ли задача или нет. В консоли увидим следующее:

```
Работа выполнена
Задача выполнена успешно
```

Использование `Task` в методах без инструкции `return` позволяет проводить отладку асинхронных методов и получать статус задачи.

3.3. Task<TResult>

Тип `Task<TResult>` используется в том случае, если асинхронный метод должен возвращать результат. При этом, под `TResult` может скрываться любой тип данных. Например, попробуем получить веб-страницу, используя асинхронный метод:

```
using System.Net.Http;
private static async Task<string> GetHtmlPage()
{
```

```
    HttpClient client = new HttpClient();
    HttpResponseMessage result = await
client.GetAsync("https://csharp.webdelphi.ru/");
    if (result.IsSuccessStatusCode) //
    {
        string data = await result.Content.ReadAsStringAsync();
        return data;
    }
    else
    {
        return "Что-то случилось";
    }
}
```

Разберем этот пример подробнее. Итак, мы создали объект типа HttpClient и во второй строке метода запускаем и ожидаем выполнения асинхронной операции:

```
HttpResponseMessage result = await
client.GetAsync("https://csharp.webdelphi.ru/");
```

Обратите внимание на то, что сам по себе метод GetAsync возвращает не HttpResponseMessage (ответ сервера), а Task<HttpResponseMessage>, но, так как применили к задаче оператор await, то получим в переменную result готовый объект типа HttpResponseMessage. Попробуйте убрать из этой строки await и получите сообщение об ошибке:

```
Ошибка CS0029 Не удается неявно преобразовать тип
«System.Threading.Tasks.Task» в
«System.Net.Http.HttpResponseMessage». ConsoleApp1
D:\YandexDisk\CSharp Sources\MathNMathF\ConsoleApp1\Program.cs
```

Далее, в условном операторе if проверяем ответ сервера и, если он успешный, то запускаем вторую асинхронную операцию – считываем ответ сервера в виде строки, после чего возвращаем результат метода через return. Опять же –не создаем объект типа Task<string>, как это, на первый взгляд, должно бы было быть – ответ асинхронной операции автоматически заворачивается в Task и уже в вызывающем методе получаем готовый Task<string>.

Вот как мог бы выглядеть метод, вызывающий нашу асинхронную операцию:

```
static async Task Main(string[] args)
{
    Task<string> downloader = GetHtmlPage();
    Console.WriteLine("Качаем страничку");
```

```
    string result = await downloader;
    Console.WriteLine(result);
}
или ещё проще:
static async Task Main(string[] args)
{
    Console.WriteLine("Качаем страничку");
    string result = await GetHtmlPage();
    Console.WriteLine(result);
}
```

В первом случае мы не применяем оператор await, предполагая, что результат нам понадобится где-то позднее, поэтому сначала получаем объект задачи. Во втором случае, сразу же ожидаем выполнения задачи и, поэтому использовали обычный тип string для получения результата.

Таким образом, использование void не рекомендуется и исключение составляет только ситуация, когда вы пишете асинхронный обработчик события (в этом случае void необходим).

4. Ожидание выполнения асинхронных задач

При разработке приложений могут возникать ситуации, когда нам необходимо не просто запустить несколько асинхронных задач и далее выполнять синхронный код, а ожидать выполнения одной или нескольких задач из списка и только затем продолжить работу. Посмотрим, как можно организовать ожидание асинхронных задач в C#.

4.1. Тестовый пример асинхронных методов

Допустим, нам необходимо с помощью трех асинхронных методов вычислить результат выражения (a+b)*c, и вывести этот результат в консоль. Очевидно, что метод вывода результата в консоль будет зависеть от результата вычислений, а метод умножения зависеть от результата суммирования двух чисел. То есть, должны получить вот такой код (если все расписывать подробно):

```
class Program
{
    //сложение двух чисел
    private static async Task<int> Sum(int a, int b)
```

```
    {
        await Task.Delay(1000);//немного подождем
        return a + b;
    }
    //умножение двух чисел
    private static async Task<int> Multiply(int a, int b)
    {
        await Task.Delay(2000);
        return a * b;
    }
    //вывод результата
    private static async Task<string> GetResult(int a)
    {
        await Task.Delay(100);
        return $"Результат равен {a}";
    }
    static async Task Main(string[] args)
    {
        int a = 0;
        int b = 3;
        int c = 4;
        int sumResult;
        int multiplyResult;
        var sum = Sum(a, b);
        Console.WriteLine("Суммирование запущено");
        //ожидаем результат сложения
        sumResult = await sum;
        var multiply = Multiply(sumResult, c);
        Console.WriteLine("Умножение запущено");
        multiplyResult = await multiply;
        //выводим результат
        Console.WriteLine(await GetResult(multiplyResult));
    }
}
```

Ожидание в методах (`Task.Delay`) добавлено, чтобы показать ИБД (имитацию бурной деятельности) приложения. Все сработает как надо. В результате должны получить `(0+3)*4=12`. Последовательно ожидаем сначала выполнения задачи `sum`, потом – `multiply` и только потом выводим в консоль результат. Такой подход вполне имеет право на существование, но, чем больше кода – тем быстрее в нем можно запутаться, да и асинхронные методы не всегда содержат всего две строчки кода, поэтому, используя возможности класса `Task` можно сделать наш пример более лаконичным и понятным.

4.2. Методы WhenAll и WhenAny

Методы `WhenAll` и `WhenAny` – это два статических метода класса `Task`. Метод `WhenAll` получает в параметрах несколько задач и создает задачу, которая будет выполнена в том случае, когда все задачи, переданные в параметрах метода будут выполнены. Этот метод подходит нам в качестве работы с примером выше. Перепишем пример, используя `WhenAll`. Пусть нам необходимо посчитать выражение: `(a+b)*(c+d)*e`:

```
static async Task Main(string[] args)
{
    int a = 0;
    int b = 3;
    int c = 4;
    int d = 5;
    int e = 6;
    //создаем две задачи
    var sum = Sum(a, b);
    var sum2 = Sum(c, d);
    //Ожидаем выполнение двух задач суммирования
    int[] results = await Task.WhenAll(sum, sum2);
    //последовательно перемножаем числа
    int multiply = await Multiply(results[0], results[1]);
    int itog = await Multiply(multiply, e);
    //выводим результат
    Console.WriteLine(await GetResult(itog));
}
```

Так как результат умножения зависит от выполнения операций сложения, то вначале создаем две задачи на сложение чисел и ожидаем пока они не обе не выполнятся. После этого последовательно перемножаем полученные результаты и выводим итог в консоль. Метод `Task.WhenAll`, так как использовали оператор `await` вернул нам массив чисел. Размерность массива совпадает с количеством параметров (отдельных задач) переданных методу. Стоит рассмотреть вопрос: что вернет метод `WhenAll`, если асинхронные задачи возвращают разные типы результатов? Например, первые две задачи должны вернуть `int`, а вторая – `string`. Тут уже явно не сможем присвоить результат `WhelAll` массиву `int[]`. Такой код вызовет ошибку:

```
//создаем две задачи
var sum = Sum(a, b);
```

```
var sum2 = Sum(c, d);
var dataPrint = GetResult(100);
//Ожидаем выполнение
int[] results = await Task.WhenAll(sum, sum2, dataPrint);
Ошибка CS0029 Не удается неявно преобразовать тип «void» в
«int[]». ConsoleApp1 Program.cs
```

В данном случае, для получения результатов нам необходимо отдельно получать результаты каждой задачи, например, так:

```
//Ожидаем выполнение
await Task.WhenAll(sum, sum2, dataPrint);
Console.WriteLine(sum.Result);
Console.WriteLine(sum2.Result);
Console.WriteLine(dataPrint.Result);
```

В отличие от `Task.WhenAll`, метод `Task.WhenAny` возвращает задачу в том случае, если хотя бы одна из задач завершится. Например:

```
var sum = Sum(a, b);
var sum2 = Sum(c, d);
Task<int> data = await Task.WhenAny(sum, sum2);
Console.WriteLine(data.Result);
```

В консоль будет выведен первый из полученных результатов. Опять же, так как обе задачи возвращают `Task<int>`, то и результатом метода `WhenAny` можно ожидать задачу `Task<int>`. Если же переданные в параметрах задачи возвращают различные типы, то результат выполнения отдельных задач нам необходимо получать через свойство `Result`:

```
await Task.WhenAny(sum, sum2, dataPrint);
if (sum.IsCompleted)
Console.WriteLine(sum.Result);
if (sum2.IsCompleted)
Console.WriteLine(sum2.Result);
if (dataPrint.IsCompleted)
Console.WriteLine(dataPrint.Result);
```

Так как в тестовом примере самым быстрым является асинхронный метод `GetResult`, то в консоли увидим результат выполнения именно этого метода.

Таким образом, для ожидания выполнения асинхронных методов можно использовать два статических метода класса `Task – WhenAll` и `WhenAny`. Метод `WhenAll` создает задачу только тогда, когда все задачи, переданные в параметрах метода, завершат

работу. WhenAny вернет результат, как только хотя бы одна из задач выполнит работу.

5. Обработка исключений в асинхронных методах

Асинхронные методы в C# также, как и другие методы могут генерировать исключения. Например, никто не даёт гарантии, что сервер вовремя ответит на ваш асинхронный запрос. Или же пользователь передаст в приложение неверные данные, например, попытается поделить число на ноль – получим ошибку. В этом случае вам может потребоваться обработка исключений в асинхронном методе.

Обработку исключений в асинхронных методах можно осуществлять также, с использованием блоков try...catch, как и в обычных методах. Однако, есть нюансы. Рассмотрим такой пример:

```
internal class Program
{
    static async Task<float> Devide(float a, float b)
    {
        await Task.Delay(100);
        if (b == 0)
            throw  new  DivideByZeroException("Делитель  равен
нулю!");
        return a / b;
    }
    static async Task Main(string[] args)
    {
        try
        {
            Console.WriteLine(await Devide(4, 0));
            Console.WriteLine(await Devide(4, 2));
        }
        catch (Exception ex)
        {
            Console.WriteLine(ex.ToString());
        }
    }
}
```

В первом ожидании результата от метода Devide должны получить ошибку – деление на ноль. Запустим приложение и убедимся, что блок catch сработал:

```
System.DivideByZeroException: Делитель равен нулю! at
ConsoleApp1.Program.Devide(Single a, Single b) in …. at
ConsoleApp1.Program.Main(String[] args) in …..
```

На первый взгляд, всё работает также, как и в синхронных методах, но, на самом деле, когда перехватывается сообщение об ошибке в асинхронном методе, то выполнение кода не прерывается и идет дальше до момента, пока не попробуем получить результат с использованием оператора `await`. Убедиться в этом просто – перепишем немного наш метод:

```
static async Task Main(string[] args)
{
    try
    {
        var res = Devide(4, 0); //ТУТ ОШИБКА
        Console.WriteLine("Выполнили ошибочное деление");
//убедимся, что выполнение других операций не прервалось
        var res2 = Devide(4, 2);
        Console.WriteLine(await res);
        Console.WriteLine(await res2);
    }
    catch (Exception ex)
    {
        Console.WriteLine(ex.ToString());
    }
    Console.ReadKey();
}
```

Теперь запустим приложение и увидим следующее:

```
Выполнили ошибочное деление System.DivideByZeroException:
Делитель равен нулю! at ConsoleApp1.Program.Devide(Single a,
Single b) in …..
```

Если запустить пошаговую отладку, то увидим следующее поведение:

```
Доходим до строки:
var res = Devide(4, 0);
Заходим в метод Devide
Генерируем исключение
Выходим из метода Devide обратно в Main
Выводим в консоль строку
Снова заходи в Devide на строке
var res2 = Devide(4, 2);
И на строке с оператором await:
Console.WriteLine(await res);
наше приложение ловит исключение и выводит его в консоль.
```

Второй момент, связанный с обработкой исключений в асинхронных методах связан с типом void. Если асинхронный метод ничего не возвращает, то исключение не передается вовне. Опять же, убедимся в этом на примере. Перепишем наш метод Devide следующим образом:

```
static async void Devide(float a, float b)
{
if (b == 0)
throw new DivideByZeroException("Делитель равен нулю!");
await Task.Delay(100);
Console.WriteLine(a / b);
}
```

Так как асинхронный метод с void ожидать нельзя, то метод Main станет таким:

```
static async Task Main(string[] args)
{
    try
    {
        Devide(4, 0); //ТУТ ОШИБКА
        Console.WriteLine("Выполнили ошибочное деление");
        Devide(4, 2);
    }
    catch (Exception ex)
    {
        Console.WriteLine(ex.ToString());
    }
    Console.ReadKey();
}
```

Теперь запустите приложение без отладки (прямо из папки bin/Debug) и убедитесь, что исключение не было отловлено в блоке catch, а вывелось в консоль как Unhandled exception.

```
Unhandled exception. Выполнили ошибочное деление
System.DivideByZeroException: Делитель равен нулю!
at ConsoleApp1.Program.Devide(Single a, Single b) in …
```

6. Где хранятся исключения асинхронных методов

Когда асинхронный метод генерирует исключение, то оно сохраняется в свойстве Task.Exception имеющим тип AggregateException, а сама задача получает статус Faulted. Рассмотрим свойство Task.Exception более подробно:

```
static async Task Main(string[] args)
{
```

```
    Task error = Devide(4, 0);
    try
    {
        await error;
    }
    catch
    {
        Console.WriteLine(error?.Status);
        Console.WriteLine(error?.Exception?.Message);

Console.WriteLine(error?.Exception?.InnerException?.Message);
    }
    Console.ReadKey();
}
```

Метод Devide возвращает объект типа Task. В консоли мы увидим следующее:

```
Faulted
One or more errors occurred. (Делитель равен нулю!)
Делитель равен нулю!
```

Наше исключение DivideByZeroException попало в свойство InnerException у Task.Exception. Дело в том, что асинхронные методы могут сгенерировать не одно, а сразу несколько исключений, например, когда мы пользуемся методом WhenAll и, чтобы иметь доступ к этим исключениям и был создан такой механизм их обработки в асинхронных методах. Вот, например, как мы можем получить сразу несколько исключений и обработать их в своем приложении:

```
Task error = Devide(4, 0);
Task error2 = Devide(5, 0);
Task error3 = Devide(6, 0);
var data = Task.WhenAll(error, error2, error3);
try
{
    await data;
}
catch
{
    Console.WriteLine(data?.Status);
    if (data?.Exception?.InnerExceptions.Count > 0)
    {
        foreach          (Exception          ex          in
data.Exception.InnerExceptions)
        {
```

```
            Console.WriteLine($"Исключение:        {ex.Message}
Источник: {ex.Source}");
        }
    }
}
```

И все они хранятся в свойстве `Task.Exception.InnerExceptions`. В консоли увидим следующее

```
Faulted
Исключение: Делитель равен нулю! Источник: ConsoleApp1
Исключение: Делитель равен нулю! Источник: ConsoleApp1
Исключение: Делитель равен нулю! Источник: ConsoleApp1
```

Таким образом, обработка исключений в асинхронных методах осуществляется также с использованием блоков `try...catch`, однако, при этом во внешнем коде исключение будет сгенерировано только в момент ожидания результата задачи (в месте использования оператора `await`). При этом, объект типа `Task` хранит сведения об исключении в свойстве `Task.Exception`. Если в нескольких ожидаемых асинхронных методах генерируются исключения, то все эти исключения попадают в свойство `Task.Exception.InnerExceptions`.

Задания и порядок выполнения работы

Задание 1. Повторить все примеры рассмотренные в теоретической части работы.

Контрольные вопросы

1. Дайте определение понятию асинхронного программирования и объясните, как оно используется в C#.

2. Что такое `Task` и как она используется для асинхронной обработки в C#?

3. Как использовать ключевое слово `async` и оператор `await` для создания асинхронных методов в C#?

4. Опишите разницу между использованием ключевого слова `async` и ключевого слова `Wait` на методах и задачах.

5. Что такое `IAsyncResult` и как он используется при асинхронном программировании в C#?

6. Объясните, что такое `SynchronizationContext` и как он связан с асинхронным программированием в C#.

7. Как управлять исключениями в асинхронных методах в C# и обрабатывать ошибки?

8. Что такое APM (Asynchronous Programming Model) и как она связана с асинхронными операциями в C#?

9. Опишите использование ключевых слов Task.Run и Task.Factory.StartNew для создания и выполнения задач в асинхронном режиме.

10. Как можно использовать библиотеки TPL (Task Parallel Library) и PLINQ (Parallel LINQ) для параллельного выполнения асинхронных операций в C#?

Лабораторная работа № 36

Тема: **«Работа с криптографическими функциями. Алгоритмы хеширования, цифровые подписи»**

Цель: изучить принципы работы с криптографическими функциями в C#, освоить алгоритмы хеширования и создание цифровых подписей для обеспечения безопасности данных и предотвращения несанкционированного доступа к информации.

Краткие теоретические сведения

Давно известно, что при взаимодействии через общедоступные сети может происходить чтение, изменение или даже кража передаваемой информации злоумышленниками. Использование различных криптографических алгоритмов позволяет обеспечить защиту данных от просмотра, обнаружить изменения в данных, а также обеспечить безопасный обмен данными на основе незащищенных каналов. Например, данные могут быть зашифрованы с помощью какого-либо криптографического алгоритма и в зашифрованном виде переданы потребителю, а затем расшифрованы потребителем, например, с помощью ключа шифрования. Если зашифрованные данные будут перехвачены злоумышленником, то расшифровать их будет трудно или практически невозможно.

В .NET C# классы для работы с шифрованием данных содержатся в пространстве имен System.Security.Cryptography. Криптография используется для достижения следующих целей:

– конфиденциальность: защита данных или личной информации пользователя от несанкционированного просмотра;

– целостность данных: защита данных от несанкционированного изменения;

– аутентификация: проверка того, что данные исходят действительно от определенного лица;

– неотрекаемость: ни одна сторона не должна иметь возможность отрицать факт отправки сообщения.

Цель хэш-алгоритма – преобразовать двоичные последовательности произвольной длины в двоичные последовательности фиксированного меньшего размера, известные

также, как хэш-значения. Хэш-значение является числовым представлением части данных. Если в хэшированном абзаце текста изменяется хотя бы одна буква, результат хэширования также меняется. Если хэш является криптостойким, его значение значительно изменится. Например, если изменяется один бит сообщения, результат выполнения криптостойкой хэш-функции может отличаться на 50%.

1. Алгоритмы хэширования в .NET C#

.NET предоставляет следующие классы, реализующие алгоритмы хэширования: SHA256, SHA384, SHA512, а также MD5 и SHA1. Но алгоритмы MD5 и SHA-1 были признаны небезопасными и сейчас рекомендуется алгоритм SHA-2, который включает в себя алгоритмы SHA256, SHA384 и SHA512. Рассмотрим, как использовать эти алгоритмы в C#.

С помощью сравнения хэш-значений можно сделать два вывода:

– сообщение не было изменено;

– отправитель сообщения подлинный

Несмотря на то, что в C# (что логично) для работы с различными алгоритмами хэширования используются также и различные классы, все они работают, по сути, по одному сценарию, который можно описать следующим образом:

– С использованием метода Create соответствующего класса создается объект (экземпляр) этого класса.

– Входное сообщение, например, строка преобразуется в массив байтов. Массив байтов передается в метод ComputeHash, который и вычисляет дайджест сообщения по заданному алгоритму и возвращает также массив байтов.

– Если необходимо преобразовать полученный массив в строку, то можно использовать, например, метод Convert.ToHexString().

Кроме этого, различные варианты метода ComputeHash могут принимать в качестве входного сообщения потоки (Stream) или же вычислять хэш для заданной части массива байтов.

2. Алгоритм MD5

MD5 (от англ. Message Digest 5) – это 128-битный алгоритм хеширования, разработанный профессором Рональдом Л. Ривестом из Массачусетского технологического института в 1991 году. Алгоритм предназначен для создания «отпечатков» или, как их ещё называют дайджестов сообщения произвольной длины и последующей проверки их подлинности. Несмотря на то, что алгоритм признан небезопасным, он до сих пор довольно часто встречается.

Рассмотрим работу с этим алгоритмом хэширования в C# на подробном примере:

```
using System.Security.Cryptography;
using System.Text;
namespace HashFunctions
{
    internal class Program
    {
        static void Main(string[] args)
        {
            string input = "Hello, World"; //входное сообщение
            string output; //дайджест сообщения (хэш)
            MD5 MD5Hash = MD5.Create(); //создаем объект для
работы с MD5
            byte[] inputBytes = Encoding.ASCII.GetBytes(input);
//преобразуем строку в массив байтов
            byte[] hash = MD5Hash.ComputeHash(inputBytes);
//получаем хэш в виде массива байтов
            output = Convert.ToHexString(hash); //преобразуем
хэш из массива в строку, состоящую из шестнадцатеричных символов
в верхнем регистре
            Console.WriteLine(output);
        }
    }
}
```

Для того, чтобы получить хэш MD5 в C# необходимо преобразовать входную строку в массив байтов и, затем, передать её в метод ComputeHash, созданного объекта типа MD5. Для преобразования полученного массива байтов мы используем метод ToHexString из класса Convert. В результате, вместо сообщения «Hello, world» мы получим в консоли следующий хэш:

```
82BB413746AEE42F89DEA2B59614F9EF
```

Следует отметить, что вне зависимости от того, какое сообщение (массив байтов) мы хэшируем, всё равно мы будем получать на выходе строку, состоящую из 32 символов (16 байт или 16*8 = 128 бит). Например, вот так выглядит хэш пустой строки ("")

```
D41D8CD98F00B204E9800998ECF8427E
```

3. Алгоритмы SHA256, SHA384 и SHA512

Этот алгоритм входит в семейство алгоритмов SHA-2. Общее описание алгоритма следующее: исходное сообщение после дополнения разбивается на блоки, каждый блок – на 16 слов. Алгоритм пропускает каждый блок сообщения через цикл с 64 (SHA256) или 80 (SHA512) итерациями (раундами). На каждой итерации 2 слова преобразуются, функцию преобразования задают остальные слова. Результаты обработки каждого блока складываются, сумма является значением хеш-функции. Тем не менее, инициализация внутреннего состояния производится результатом обработки предыдущего блока. Поэтому независимо обрабатывать блоки и складывать результаты нельзя.

Этот алгоритм может использоваться для проверки подлинности сообщения. Даже малейшее изменение исходного сообщения приводит к тому, что хэш значительно изменяется. Например, вычислим хэш двух строк, которые различаются всего одним символом. Метод, в котором получаем хэш SHA256:

```
using System.Security.Cryptography;
using System.Text;
namespace HashFunctions
{
    internal class Program
    {
        public static string CreateSHA256(string input)
        {
            using SHA256 hash = SHA256.Create();
            return
Convert.ToHexString(hash.ComputeHash(Encoding.ASCII.GetBytes(i
nput)));
        }

        static void Main(string[] args)
```

```
        {
            Console.WriteLine(CreateSHA256("Hello, world"));
            Console.WriteLine(CreateSHA256("Hello. world"));

        }
    }
}
```

Результат вычислений:

```
4AE7C3B6AC0BEFF671EFA8CF57386151C06E58CA53A78D83F36107316CEC125F
143D348D0565AF587AADCE8DB4C3891CF6CF8BD7863113C82766E9B3F6AB821D
```

Аналогичным образом действуют и алгоритмы SHA384 и SHA512, но, соответственно хэш получится в два раза больше, например, ниже представлены результаты вычисления хэша для строки «Привет, мир» с использованием алгоритма SHA256 и SHA512:

```
SHA256:
2A2E76364DF5AB8F0441D9C88BF7688F7F565F0F6B92A877CC94263E123021
E3
SHA512:
39B5FA93892703D4C95BC8C16D42599B988B83602AB1775849FC3C424D46F4
0B791484A9E518DF027ACBA901ED3739CBFE7D40BF7FBB0117E6B6047457C7
F864
```

4. Алгоритмы хэширования и русские символы

Довольно часто, в начале работы с различными хэш-алгоритмами можно столкнуться с такой проблемой – хэш, созданный вами в C# не совпадает с тем хэшем, который вы получили, например, от какого-либо онлайн-сервиса. В большинстве случаев, проблема кроется в кодировке текста. Например, вернемся к алгоритму SHA256. Как было сказано выше, малейшее изменение сообщения должно привести к значительной смене результата. Возьмем два русских слова – «кот» и «код» и попробуем вычислить их хэш с помощью следующего метода:

```
public static string CreateSHA256(string input)
{
using SHA256 hash = SHA256.Create();
return
Convert.ToHexString(hash.ComputeHash(Encoding.ASCII.GetBytes(i
nput)));
}
```

В результате получим неверное значение хэшей. Более того, изменение одного символа вообще никак не затронуло значение хэша:

```
A03B221C6C6EAE7122CA51695D456D5222E524889136394944B2F9763B483615
A03B221C6C6EAE7122CA51695D456D5222E524889136394944B2F9763B483615
```

Проблема кроется в кодировке символов. Мы использовали в этом примере кодировку ASCII, в то время как должны были использовать Unicode. Для получения верного результата перепишем пример следующим образом:

```
public static string CreateSHA256(string input)
{
using SHA256 hash = SHA256.Create();
return
Convert.ToHexString(hash.ComputeHash(Encoding.UTF8.GetBytes(in
put)));
}
```

Теперь хэши будут действительно сильно различаться:

```
77095EBDD157B8410550B35798F39C5C0C6085580F5F35777865FD4179AA2599
87FC5157A761FC742B0258FE3753A062257288F65F2C6BF74200B4829B14DF95
```

Почему именно UTF-8, а не UTF-16 или UTF-32? Можно использовать и эти кодировки для русских символов, т.к. всё это Unicode, и результат не изменится для русского текста. Но, всё же в Сети на данный момент, преобладает кодировка UTF-8 и потому лучше использовать именно её, чтобы с большей вероятностью получать переносимый результат, который будет принят каким-либо внешним сервисом.

5. Алгоритмы подписи данных в .NET C#. HMAC

HMAC (от англ. hash-based message authentication code, код проверки подлинности сообщений, использующий односторонние хеш-функции) – один из механизмов проверки целостности информации, позволяющий гарантировать то, что данные, передаваемые или хранящиеся в ненадёжной среде, не были изменены злоумышленниками.

HMAC смешивает секретный ключ с данными сообщения, хэширует результат с хэш-функцией, снова смешивает хэш-значение с секретным ключом, а затем применяет хэш-функцию во второй раз. HMAC можно использовать для определения того, что при отправке сообщения через небезопасный канал оно не было

изменено при условии, что отправитель и получатель совместно используют секретный ключ. Отправитель вычисляет хэш-значение для исходных данных и отправляет исходные данные и хэш-значение в виде одного сообщения. Получатель пересчитывает хэш-значение полученного сообщения и проверяет, совпадают ли хэши. Таким образом, если исходные и вычисляемые хэш-значения совпадают, сообщение проходит проверку подлинности.

В C# реализованы следующие алгоритмы для подписи и проверки подлинности данных: HMAC-MD5, HMAC-SHA1, HMAC-SHA256, HMAC-SHA384 и HMAC-SHA512. Как и в случае с хэш-алгоритмами, все алгоритмы HMAC работают также по одним и тем же принципам, поэтому, рассмотрим работу с ними на примере, наверное, самого популярного алгоритма подписи данных в настоящее время – HMAC-SHA256.

Подписываем данные, используя HMAC-SHA256. Для демонстрации, создадим рядом с exe-файлом приложения файл с названием secretfile.txt в который запишем одну строку – «Пример работы HMAC-SHA256». Напишем приложение, которое подпишет этот файл. Также, для работы нам потребуется секретный ключ. Пусть это будет «password».

Следующий код демонстрирует пример подписи файла:

```
using System.Security.Cryptography;
using System.Text;

namespace HashFunctions
{
    internal class Program
    {
        static void Main(string[] args)
        {
            string sourceFileName = "secretfile.txt";
            string outputFileName = "secretfile.enc";
            string key = "password";
            using HMACSHA256 hmac = new
HMACSHA256(Encoding.UTF8.GetBytes(key)); //создаем объект для
работы и передаем в него ключ
            using FileStream inStream = new
FileStream(sourceFileName, FileMode.Open); //создаем файловый
поток на чтение
            Console.WriteLine(inStream);
```

```
            using FileStream outStream = new
FileStream(outputFileName, FileMode.Create);//создаем файловый
поток на запись
            byte[] hashData = hmac.ComputeHash(inStream);
//вычисляем хэш
            inStream.Position = 0; //возвращаемся в начало
потока, содержащего исходный файл
            outStream.Write(hashData, 0,
hashData.Length);//пишем в выходной поток полученный хэш

//копируем данные из исходного файла в подписываемый
            int bytesRead;
            byte[] buffer = new byte[1024];
            do
            {
                bytesRead = inStream.Read(buffer, 0, 1024);
                outStream.Write(buffer, 0, bytesRead);
            } while (bytesRead > 0);
        }
    }
}
```

Что получится в итоге? Вот так выглядит исходный файл (рис. 36.1):

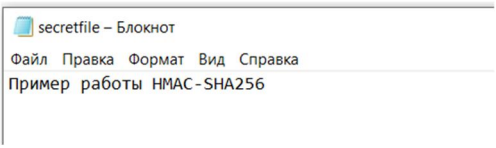

Рис. 36.1. Текст для подписи

Вот так будет выглядеть выходной файл (secretfile.enc) (рис. 36.2):

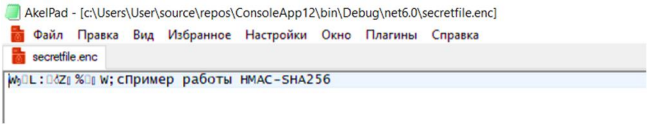

Рис. 36.2. HMAC-SHA256 файл

6. Проверка подписи

Перейдем к следующему шагу – реализуем проверку подписанного файла, чтобы убедиться, что данные не были изменены. Ниже представлен метод проверки подписи:

```
public static bool VerifyFile(byte[] key, string sourceFile)
        {
            bool err = false;
            using (HMACSHA256 hmac = new HMACSHA256(key))
            {
                byte[] storedHash = new byte[hmac.HashSize / 8]; //массив для сохранения хэша из файла (32 байта = 256 / 8)
                using (FileStream inStream = new FileStream(sourceFile, FileMode.Open)) //теперь исходный файл - это файл с подписью
                {
                    inStream.Read(storedHash, 0, storedHash.Length); //читаем подпись после этой операции курсор в потоке будет находиться ровно в начале содержимого
                    byte[] computedHash = hmac.ComputeHash(inStream); //считаем хэш оставшегося содержимого (исключая подпись).
                    err = computedHash.Except(storedHash).Any(); //если, хотя бы по одному байту не совпали - файл подделали
                }
            }
            if (err)
            {
                Console.WriteLine("Значения хеша отличаются! Подписанный файл был изменен!");
                return false;
            }
            else
            {
                Console.WriteLine("Хэш-значения совпадают — файл настоящий.");
                return true;
            }
        }
```

То есть, в двух словах, суть проверки подписи следующая: так как мы вставляли подпись в файл с начала, то и при проверке мы считываем подпись от начала файла. Дополнительно никаких хэшей считать не требуется. Далее мы считаем хэш, используя наш секретный ключ, для оставшейся части файла, то есть того, что подписывалось. И полученный хэш побайтово сравниваем с тем, который «вырезали» из начала файла. Чтобы не писать лишний цикл, можно воспользоватся методом LINQ и проверить есть ли хотя бы одно расхождение в двух массивах. Если расхождений нет – значит подписи совпали и файл настоящий.

Ну и остается последний момент – как вернуть файл в исходное состояние. Всё достаточно просто – необходимо скопировать, например, в другой поток всё содержимое после подписи. Например, следующим образом:

```
using System.IO;

inStream.Position = storedHash.Length; //поток с подписанным
файлом. Смещаемся на позицию ЗА подпись
using FileStream outStream = new FileStream("decrypt.txt",
FileMode.Create); //создаем новый поток для записи контента
int bytesRead; //читаем содержимое из одного потока в другой
byte[] buffer = new byte[1024];
do
{
    bytesRead = inStream.Read(buffer, 0, 1024);
    outStream.Write(buffer, 0, bytesRead);
} while (bytesRead > 0);
```

В результате получим исходный файл.

При разработке Windows Forms приложений удобно использовать диалоговые окна выбора файла. Для этого используется класс OpenFileDialog. Для примера, создадим проект Windows Forms приложения (см. рис. 36.3) и добавим на форму одну кнопку и компонент OpenFileDialog из группы компонентов – Диалоговые окна (рис. 36.3).

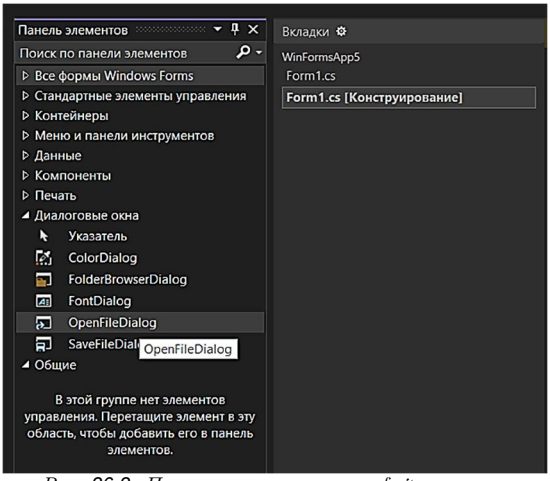

Рис. 36.3. Проектирование интерфейса

Создадим следующий обработчик события нажатия кнопки

```
private void button1_Click(object sender, EventArgs e)
        {

            if         (openFileDialog1.ShowDialog()         ==
DialogResult.OK)
            {
                MessageBox.Show(openFileDialog1.FileName);
            }

        }
```

Данный пример иллюстрирует как получить имя выбранного пользователем файла. Для фильтрации выбираемых файлов можно использовать свойство Filter:

```
openFileDialog1.Filter = "Файлы txt|*.txt|Файлы cs|*.cs";
```

В результате, при выборе файла будут доступны только форматы txt и cs (рис. 36.4).

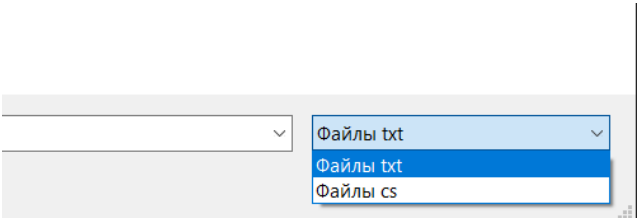

Рис. 36.4. Фильтрация по типам файлов

Задания и порядок выполнения работы

Задание 1. Выполнить все примеры из теоретической части.

Задание 2. Найти хэш-значение своей фамилии с помощью алгоритмов MD5, SHA256, SHA512. В консоль должна выводиться и фамилия и ее хэш, с указанием метода хэширования.

Задание 3. Внести изменение в текст подписанного HMACSHA256 файла и выполнить проверку. Убедиться, что проверка не была пройдена.

Задание 4. Разработать Windows Forms приложение, которое содержит одно поле – для ввода произвольной текстовой строки, а также поля для вывода данных. На форме должна находится кнопка, по нажатию которой происходит вычисление

хэш-значения по алгоритмам MD5, SHA256, SHA512 и выводится на форму.

Задание 5. Разработать Windows Forms приложение, которое содержит поле для ввода секретного ключа, а также позволяет пользователю выбрать произвольный текстовый файл (выбор файла осуществляется с помощью диалогового окна выбора файла, с фильтром для формата txt) для подписи этого файла с помощью алгоритма HMACSHA256. Подписанный файл, должен сохраняться рядом с файлом exe разработанного приложения.

Контрольные вопросы

1. Что изучает криптография, объясните ее роль в информационных технологиях.
2. Перечислите основные задачи, которые решаются с помощью криптографии.
3. Что такое хеширование и зачем оно используется?
4. В чем разница между хешированием и шифрованием?
5. Какие алгоритмы хеширования вы знаете и в чем их особенности?
6. Что такое цифровая подпись и как она используется для обеспечения целостности данных?
7. Какие типы атак на цифровые подписи вы знаете и как их можно предотвратить?
8. Опишите процесс создания цифровой подписи с использованием асимметричного шифрования.
9. В чем отличие между симметричным и асимметричным шифрованием при создании цифровых подписей?
10. Какие классы и методы в C# используются для работы с криптографическими функциями?

Лабораторная работа № 37

Тема: **«Универсальные шаблоны. Очереди, стеки, словари»**

Цель: изучить принципы работы с универсальными шаблонами в C#, а также такие структуры данных как очереди, стеки, словари.

Краткие теоретические сведения

В языке программирования C#, универсальные шаблоны представлены в виде обобщений (generics). Обобщения в C# позволяют создавать классы, интерфейсы, методы и делегаты, которые могут работать с разными типами данных, сохраняя при этом безопасность типов. Они предоставляют возможность создавать компоненты программы, которые могут быть параметризованы типами данных, а не привязаны к конкретным типам в процессе написания кода.

Создадим класс, который будет содержать массив любого типа данных.

```
class ArrayClass<T>
{
    T[] data;
    uint index = 0;

    public ArrayClass(uint length)
    {
        data = new T[length];
    }

    public bool Add(T item)
    {
        if (index > data.Length)
            return false;
        data[index++] = item;
        return true;
    }

    public T Get(uint index)
    {
        return (index < this.index) && (index >= 0) ?
data[index] : default(T);
    }
```

```
    public uint Count()
    {
        return index;
    }
}
```

Посмотрим, чем универсальный класс ArrayClass отличается от обычных классов C#. Во-первых, рядом с названием класса появились треугольные скобки с символом T внутри. Скобки <> указывают на то, что класс является универсальным, а тип T, заключенный в угловые скобки, будет использоваться этим классом. В принципе, вместо буквы T мы могли бы использовать другие буквы, но по умолчанию принято указывать универсальный тип как T (видимо, от слова Template – шаблон). При этом на этапе создания класса мы не знаем какой тип данных будет использоваться в классе, поэтому параметр T в угловых скобках также называется универсальным параметром, так как вместо него можно подставить любой тип.

Посмотрим на примеры использования нашего класса:

```
ArrayClass<int> intArray = new ArrayClass<int>(10);
intArray.Add(1);
intArray.Add(2);
intArray.Add(3);
intArray.Add(4);
intArray.Add(5);
Console.WriteLine(intArray.Count());//5
Console.WriteLine(intArray.Get(2));//2
Console.WriteLine(intArray.Get(100));//0
```

Создали класс, который будет содержать массив целых чисел. Так как наш класс универсальный, то мы можем без проблем создать и класс с массивом строк:

```
ArrayClass<string> strArray = new ArrayClass<string>(10);
strArray.Add("1");
strArray.Add("2");
strArray.Add("3");
strArray.Add("4");
Console.WriteLine(intArray.Count());//4
Console.WriteLine(intArray.Get(1));//1
Console.WriteLine(intArray.Get(500));//0
```

1. Очереди

Очередь – это структура данных, содержащая один или несколько элементов одного типа и работающая по принципу FIFO («first in – first out», «первый вошел – первый вышел»). В отличие от списка или массива, очередь не допускает произвольный доступ к своим элементам – чтение происходит только с первого элемента по порядку до конца. Так же, мы не можем поместить новый элемент в произвольную позицию очереди – только в конец очереди.

Класс `Queue<T>` представляет собой очередь в C#. Рассмотрим основные моменты работы с очередью в C#.

Использование очереди удобно в том случае, когда вам необходимо создать временное хранилище элементов, то есть, когда вам необходимо удалять элемент после получения его значения. Конечно, и в этом случае, мы можем создать обычный список `List<T>`, организовав получение значения в определенном порядке и удаление элемента из списка, но, используя очередь `Queue<T>` мы экономим время на реализацию.

Все мы прекрасно знаем, как работают различные электронные очереди в банках, больницах и т.д. Попробуем реализовать электронную очередь в C#. То есть, нам необходимо сделать так, чтобы первый, поставленный в очередь элемент (клиент) первым из этой очереди и вызывался.

```
//создаем очередь клиентов
Queue<string> clients = new Queue<string>();
//добавляем в очередь элементы
clients.Enqueue("A1");
clients.Enqueue("A2");
clients.Enqueue("B1");
clients.Enqueue("B2");

//создаем очередь свободных окон
Queue<string> windows = new Queue<string>(2);
windows.Enqueue("Окно 1");
windows.Enqueue("Окно 2");

//вызываем клиентов. Первый вошел - первый вышел
while (clients.Count > 0)
{
    string client = clients.Dequeue();
```

```
    if (windows.Count > 0)
        Console.WriteLine($"Клиент         {client}         -->
{windows.Dequeue()}");
    else
    {
        Console.WriteLine($"Клиент {client} остается ждать");
    }
}
```

Создается две очереди:

– Очередь клиентов (clients)

– Очередь свободных окон (windows)

Используя метод Enqueue, в очередь клиентов добавляются четыре клиента, а в очередь окон два свободных окна. Обратите внимание на создание очереди windows:

```
Queue<string> windows = new Queue<string>(2);
```

В круглых скобках мы указываем начальную емкость хранилища – два элемента.

Далее, используя метод Dequeue мы «вызываем» клиентов по очереди к свободным окнам. Если все окна заняты, то выводим сообщение о том, что клиент остается в очереди ожидания. Результат выполнения программы будет следующим:

```
Клиент A1 -> Окно 1
Клиент A2 -> Окно 2
Клиент B1 остается ждать
Клиент B2 остается ждать
```

Основные методы Queue представлены ниже:

- Clear() – удаляет все объекты из Queue<T>.

- Contains(T) – определяет, входит ли элемент в коллекцию Queue<T>.

- CopyTo(T[], Int32) – копирует элементы коллекции Queue<T> в существующий одномерный массив Array, начиная с указанного значения индекса массива.

- Dequeue() – удаляет объект из начала очереди Queue<T> и возвращает его.

- Enqueue(T) – добавляет объект в конец коллекции Queue<T>.

- Peek() – возвращает объект, находящийся в начале очереди Queue<T>, но не удаляет его.

- ToArray() – копирует элементы Queue<T> в новый массив.

- TrimExcess() – устанавливает емкость равной фактическому количеству элементов в Queue<T>, если это количество составляет менее 90 процентов текущей емкости.

- TryDequeue(T) – удаляет объект в начале Queue<T> и копирует его в параметр result.

- TryPeek(T) – возвращает значение, указывающее, имеется ли в начале Queue<T> объект, и, если он присутствует, копирует его в параметр result. Объект не удаляется из Queue<T>.

Стоит обратить внимание на следующий момент: при попытке получить элемент из очереди, используя методы Enqueue или Peek может возникнуть исключение, если очередь будет пуста.

Чтобы избежать исключения мы можем воспользоваться методами TryDequeue или TryPeek или же проверить количество элементов перед вызовом методов, например, так как мы это делали в примере выше:

```
if (windows.Count > 0)
    Console.WriteLine($"Клиент {client} -->
{windows.Dequeue()}");
else
{
    Console.WriteLine($"Клиент {client} остается ждать");
}
```

2. Стеки

Стек – это структура данных (коллекция, последовательность), содержащая один или несколько элементов одного типа и работающая по принципу LIFO («last in – first out», «последний вошел – первый вышел»).

В C# стек представляет класс обобщенный класс Stack<T>. Вывод элементов стека происходит в обратном порядке (принцип LIFO), в то время как используя очередь (Queue) вывод элементов происходит в прямом порядке (первый вошел – первый вышел), а, используя список List, мы можем организовать вывод в любом удобном для нас порядке.

Для создания стека в C# мы можем воспользоваться различными конструкторами.

Создание пустого стека:

```
Stack<int> stack = new Stack<int>();
```

Создание пустого стека заданной емкости:

```
Stack<int> stack2 = new Stack<int>(10);
```

Создали стек с начальной емкостью в десять элементов.

Создание стека из коллекции:

```
int[] array = new int[] { 1, 2, 3, 4, 5 };
Stack<int> stack3 = new Stack<int>(array);
```

Зесь мы создали стек, используя обычный одномерный массив. В этот конструктор можно передавать любые объекты, реализующие интерфейс IEnumerable.

- Clear() – удаляет все объекты из Stack<T>.

- Contains(T) – определяет, входит ли элемент в коллекцию Stack<T>.

- CopyTo(T[], Int32) – копирует Stack<T> в существующий одномерный массив Array, начиная с указанного индекса массива.

- Peek() – возвращает объект, находящийся в начале Stack<T>, без его удаления.

- Pop() - удаляет и возвращает объект, находящийся в начале Stack< T>.

- Push(T) – вставляет объект как верхний элемент стека Stack<T>.

- ToArray() – копирует Stack<T> в новый массив.

- TrimExcess() – устанавливает емкость равной фактическому количеству элементов в Stack<T>, если это количество составляет менее 90 процентов текущей емкости.

Рассмотрим простой пример работы со стеком в C#, показывающий использование различных методов класса Stack<T>:

```
using System;
using System.Collections.Generic;

namespace ConsoleApp7
{
    internal class Program
    {
        static void Main(string[] args)
        {
            Stack<int> stack = new Stack<int>(3); //создаем
стек с начальной емкостью на 3 элемента
```

```
            Console.WriteLine($"Количество элементов коллекции
{stack.Count}");

            //добавляем новые элементы в стек
            stack.Push(1);
            stack.Push(2);
            stack.Push(3);
            stack.Push(4);//на  этом  шаге  емкость  хранилища
будет увеличена
            stack.Push(5);
            while (stack.Count > 0)
            {
                if (stack.Peek() == 3)
                {
                    Console.WriteLine($"Нашли в стеке значение
3. Всего элементов {stack.Count}");
                }
                Console.WriteLine($"Извлекли из стека значение
{stack.Pop()}. Всего элементов {stack.Count}");
            }
        }
    }
}
```

Результат работы приложения будет следующий:

```
Количество элементов коллекции 0
Извлекли из стека значение 5. Всего элементов 4
Извлекли из стека значение 4. Всего элементов 3
Нашли в стеке значение 3. Всего элементов 3
Извлекли из стека значение 3. Всего элементов 2
Извлекли из стека значение 2. Всего элементов 1
Извлекли из стека значение 1. Всего элементов 0
```

3. Словари

Коллекция `Dictionary<K,V>` представляет собой так называемый словарь. В такой коллекции каждый элемент представляет собой пару «ключ – значение». Соответственно, основное назначение `Dictionary<K,V>` – это получение значения по его ключу. По сути, словарь – это хэш-таблица и получение значения с помощью ключа выполняется очень быстро, близко к O (1). Скорость извлечения значения (V) зависит от качества алгоритма хэширования для типа, указанного для ключа(K).

Класс `Dictionary` предоставляет сразу восемь различных конструкторов для создания словаря. Рассмотрим основные способы создания словаря и его наполнения данными.

Создание пустого словаря:

```
Dictionary<int, string> map = new Dictionary<int, string>();
```

Создали пустой словарь, у которого в качестве ключа выступает целое число (`int`), а в качестве значения – строка (`string`). Теперь мы можем наполнить наш словарь данными, например, так:

```
map.Add(1, "Яблоко");
map.Add(2, "Апельсин");
map.Add(3, "Груша");
```

При этом, стоит обратить внимание на то, что ключ – это обязательно уникальное значение. Например, следующий код вызовет исключение:

```
map.Add(1, "Яблоко");
map.Add(1, "Апельсин");
System.ArgumentException: «An item with the same key has already
been added. Key: 1»
```

Создание и инициализация словаря значениями:

```
Dictionary<int, string> map = new Dictionary<int, string>()
{
    { 1, "Яблоко"},
    { 2, "Апельсин"},
    { 3, "Груша"}
};
```

В фигурных скобках мы указываем пары «ключ–значение». Так как ключом у нас выступает тип `int`, то, соответственно, и при наполнении словаря, вначале идёт ключ, а, затем, значение элемента.

Также, аналогичный вариант инициализации:

```
Dictionary<int, string> map = new Dictionary<int, string>()
{
    [1] = "Яблоко",
    [2] = "Апельсин",
    [3] = "Груша"
};
```

В квадратных скобках указывается ключ, а далее, после знака равенства (`=`) – значение.

4. Перебор элементов в цикле foreach

Самый простой способ перебора элементов словаря – в цикле `foreach`:

```
foreach (var fruit in map)
{
    Console.WriteLine($"Ключ {fruit.Key} значение
{fruit.Value}");
}
```

При этом каждый элемент словаря будет представлять из себя экземпляр структуры `KeyValuePair<TKey, TValue>` или, если быть точным, в нашем примере – это `KeyValuePair<int, string>`. После того, как мы получили очередной элемент словаря, мы можем получить значения его полей `Key` и `Value`.

Все значения ключей словаря содержатся в свойстве `KeyCollection`. В примере ниже показано, как можно перебрать все значения ключей словаря:

```
Dictionary<int, string> map = new Dictionary<int, string>()
{
    [1] = "Яблоко",
    [2] = "Апельсин",
    [3] = "Груша"
};
//получае коллекцию ключей
Dictionary<int, string>.KeyCollection keys = map.Keys;
Console.WriteLine($"Количество ключей {keys.Count}");
//перебираем значения ключей
foreach (int key in keys)
{
    Console.WriteLine(key);
}
```

Аналогичным образом можно получить доступ к коллекции значений и перебрать их в цикле:

```
Dictionary<int, string>.ValueCollection values = map.Values;
foreach (string value in values)
{
    Console.WriteLine(value);
}
```

5. Доступ к элементам словаря по значению ключа

Рассмотренные выше примеры работы с элементами словаря не раскрывают основного назначения этого типа данных, а именно – получение значения по ключу. Рассмотрим следующий пример:

```
Dictionary<int, string> map = new Dictionary<int, string>()
{
    [1] = "Яблоко",
    [2] = "Апельсин",
    [3] = "Груша"
};
for (int i = 0; i < 5; i++)
{
   string value = map.GetValueOrDefault(i);
   if (value == null)
        Console.WriteLine($"В     словаре     нет     значения
соответствующего ключу {i}");
   else
        Console.WriteLine(value);
}
```

В результате, в консоли мы увидим следующий вывод:

```
В словаре нет значения соответствующего ключу 0
Яблоко
Апельсин
Груша
В словаре нет значения соответствующего ключу 4
```

Этот же пример мы могли бы переписать и следующим образом:

```
for (int i = 0; i < 5; i++)
{
   if (map.TryGetValue(i, out string value))
        Console.WriteLine(value);
   else
        Console.WriteLine($"В     словаре     нет     значения
соответствующего ключу {i}");
}
```

Результат будет тот же, что и в примере выше.

Ниже представлены основные методы, реализованные в классе Dictionary

- `Add(TKey, TValue)` – добавляет указанные ключ и значение в словарь.

- `Clear()` – удаляет все ключи и значения из словаря `Dictionary<TKey,TValue>`.

- `ContainsKey(TKey)` – определяет, содержится ли указанный ключ в словаре `Dictionary<TKey,TValue>`.

- `ContainsValue(TValue)` – определяет, содержит ли коллекция `Dictionary<TKey,TValue>` указанное значение.

- `Remove(TKey)` – удаляет значение с указанным ключом из словаря `Dictionary<TKey,TValue>`.

- `TryGetValue(TKey, TValue)` – получает значение, связанное с заданным ключом.

Задания и порядок выполнения работы

Задание 1. Создайте систему управления очередью заявок. Реализуйте класс «Очередь заявок», который позволяет добавлять заявки и обрабатывать их в порядке поступления. Каждая заявка должна содержать информацию о времени поступления и описании. Для добавления заявок и их обработки использовать графический интерфейс.

Задание 2. Напишите программу, которая считывает текстовый документ и использует словарь для подсчета частоты встречаемости каждого слова. Выведите на экран список слов и соответствующих им частот.

Задание 3. Создайте простую программу с графическим интерфейсом, которая позволяет пользователю вводить текстовые данные и изменять их. Используйте стек для реализации функции «отката» (undo), которая позволит отменять последние внесенные изменения в текст.

Контрольные вопросы

1. В чем разница между очередью и стеком в C#?
2. Как создать и использовать стек в C#?
3. Как добавить и удалить элементы из стека в C#?
4. В чем преимущества и недостатки использования стека по сравнению с другими структурами данных?
5. Как создать словарь в C# и какие типы данных могут быть его ключами и значениями?
6. Как добавить, удалить и получить элемент из словаря в C#?
7. Как работает индексатор в словаре в C#? В чем его отличие от метода `ContainsKey()`?
8. Какие особенности и ограничения есть у словарей с точки зрения производительности?

9. Можно ли изменить ключ или значение элемента в словаре после его добавления? Если да, то как это сделать?

10. Что такое «словари с разными типами ключей» в C# и как их использовать?

Лабораторная работа № 38

Тема: **«Парсинг веб-страниц с помощью C#»**

Цель: изучить и реализовать на практике различные методы парсинга веб-страниц с использованием языка программирования C#, а также проанализировать возможные проблемы и ограничения, связанные с парсингом HTML-кода страниц.

Краткие теоретические сведения

HtmlAgilityPack (HAP) – это библиотека для .NET, предназначенная для работы с HTML-документами. Она предоставляет удобные средства для парсинга HTML, навигации по структуре документа и выполнения различных операций над HTML-элементами. HtmlAgilityPack позволяет разработчикам извлекать данные из HTML-страниц, манипулировать DOM-деревом и взаимодействовать с содержимым в удобной форме.

HtmlAgilityPack позволяет загружать HTML-документы из различных источников, таких как строки, файлы или веб-ресурсы. Библиотека поддерживает использование XPath-выражений для выбора и фильтрации элементов в HTML-документе, что делает навигацию и поиск данных более гибкими.

Разработчики могут добавлять, удалять и изменять HTML-элементы, а также их атрибуты, что обеспечивает возможность внесения изменений в структуру документа.

Для использования HtmlAgilityPack в проекте на C#, необходимо установить пакет с помощью NuGet Package Manager (рис. 38.1):

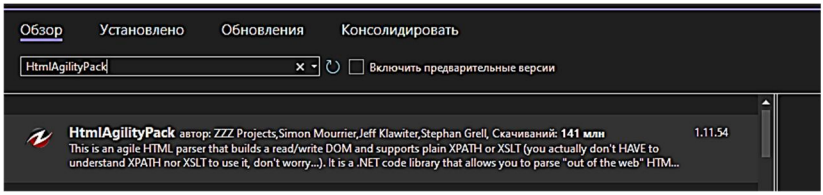

Рис. 38.1. Установка HtmlAgilityPack

Пример 1. Приведем пример загрузки HTML-страницы и выбора всех заголовков <h1>:

```
using HtmlAgilityPack;
```

```
using System;

class Program
{
    static void Main()
    {
        var html = "<html><body><h1>Заголовок
1</h1><h1>Заголовок 2</h1></body></html>";
        var document = new HtmlDocument();
        document.LoadHtml(html);

        var headers =
document.DocumentNode.SelectNodes("//h1");

        if (headers != null)
        {
            foreach (var header in headers)
            {
                Console.WriteLine("Заголовок: " +
header.InnerText);
            }
        }
    }
}
```

Пример 2. Загрузка содержимого веб-страницы

```
using HtmlAgilityPack;
using System;

class Program
{
    static void Main()
    {
        // URL страницы
        string url = "https://lgpu.org/news/?page=1";

        // Создаем объект HtmlWeb
        var web = new HtmlWeb();

        try
        {
            // Загружаем HTML-страницу
            var document = web.Load(url);

            // Выводим HTML-код страницы
            Console.WriteLine(document.Text);
        }
        catch (Exception ex)
```

```
        {
            Console.WriteLine("Произошла ошибка при загрузке
страницы: " + ex.Message);
        }
    }
}
```

Пример 3. Поиск элементов по названию класса

```
using System;
using HtmlAgilityPack;
using System.Collections.Generic;
using System.Linq;

public class Program
{
    public static void Main()
    {
        // declaring & loading dom
        HtmlWeb web = new HtmlWeb();
        HtmlAgilityPack.HtmlDocument doc = new
HtmlAgilityPack.HtmlDocument();
        doc = web.Load("https://lgpu.org/");

        IEnumerable<HtmlNode> nodes =
doc.DocumentNode.Descendants().Where(n =>
n.HasClass("title"));

        foreach (var item in nodes)
        {
            // displaying final output
            Console.WriteLine(item.InnerText);
        }
    }
}
```

Пример 4. Получение метаданных

```
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using HtmlAgilityPack;

static void GetMetaInformation(HtmlAgilityPack.HtmlDocument
htmldoc, string value)
{
    HtmlNode tcNode =
htmldoc.DocumentNode.SelectSingleNode("//meta[@name='" + value
+ "']");
```

```
    string fulldescription = string.Empty;
    if (tcNode != null)
    {
        HtmlAttribute desc;
        desc = tcNode.Attributes["content"];
        Console.ForegroundColor = ConsoleColor.Red;
        Console.Write(desc.Value);
        Console.ReadLine();
    }
}

HtmlWeb web = new HtmlWeb();
HtmlAgilityPack.HtmlDocument doc = new
HtmlAgilityPack.HtmlDocument();
doc = web.Load("https://lgpu.org/");
GetMetaInformation(doc, "description");
```

Задания и порядок выполнения работы

Задание 1. Выполните парсинг главной страницы сайта. С помощью HtmlAgilityPack получите исходный код главной страницы сайта https://lgpu.org/.

Задание 2. Извлеките все заголовки тексты страницы https://lgpu.org/news/?page=1 и выведите их в консоль.

Задание 3. Извлеките все заголовки новостей со страниц 1-10 (https://lgpu.org/news/?page=1) и выведите их в консоль.

Задание 4. Извлеките все ссылки со страницы https://lgpu.org/ В качестве примера можно использовать код:

```
using HtmlAgilityPack;
using System;

class Program
{
    static void Main()
    {
        var html = "<html><body><a
href='https://example.com'>Ссылка 1</a><a
href='https://example.org'>Ссылка 2</a></body></html>";
        var document = new HtmlDocument();
        document.LoadHtml(html);

        var links = document.DocumentNode.SelectNodes("//a");

        if (links != null)
        {
```

```
            foreach (var link in links)
            {
                Console.WriteLine("Ссылка: " +
link.GetAttributeValue("href", ""));
            }
        }
    }
}
```

Задание 5. Извлеките все изображения с веб-страницы https://lgpu.org/

```
using HtmlAgilityPack;
using System;

class Program
{
    static void Main()
    {
        var html = "<html><body><img src='image1.jpg'
alt='Изображение 1'><img src='image2.jpg' alt='Изображение
2'></body></html>";
        var document = new HtmlDocument();
        document.LoadHtml(html);

        var images =
document.DocumentNode.SelectNodes("//img");

        if (images != null)
        {
            foreach (var image in images)
            {
                Console.WriteLine("Изображение: " +
image.GetAttributeValue("src", ""));
                Console.WriteLine("Alt текст: " +
image.GetAttributeValue("alt", ""));
            }
        }
    }
}
```

Контрольные вопросы

1. Дайте определение термину «парсинг веб-страниц».
2. Какие основные методы парсинга веб-страниц вы знаете?
3. В чем разница между синхронным и асинхронным парсингом веб-страниц?

4. Какие особенности необходимо учитывать при работе с HTML-кодом страниц при их парсинге?

5. Что такое регулярные выражения и как они используются в парсинге веб-страниц на C#?

6. Что такое XPath и как он используется для поиска информации на веб-страницах?

7. Как обрабатывать и анализировать данные, полученные в результате парсинга веб-страницы?

8. Какие основные проблемы могут возникнуть при парсинге веб-страниц и как их можно избежать?

9. Какие инструменты и библиотеки используются для парсинга веб-страниц в C#?

10. Как обеспечить безопасность и анонимность при парсинге веб-сайтов?

Заключение

В заключительной части лабораторного практикума представлены лабораторные работы, направленные на дальнейшее более углубленное изучение синтаксиса языка C#.

Третья часть посвящена разработке приложений с использованием современных технологий параллельного, многопоточного и асинхронного программирования. Также в лабораторных работах изучались основные методы и подходы работы с базами данных. В частности, рассмотрены как файловые базы данных, так и клиент-серверные базы данных.

Выполнение лабораторных работ данного практикума позволит получить навыки работы с датой и временем в C#, приобрести практические навыки по работе с криптографическими примитивами и хэш-функциями, изучить возможности работы с форматом JSON выполняя сериализацию и десериализацию различных данных в классах.

Авторы убеждены, что данный лабораторный практикум станет полезным инструментом в обучении студентов основам программирования на платформе .Net, и желают успехов в освоении этого увлекательного и перспективного направления информационных технологий.

Список рекомендуемой литературы

1. Васильев, А. Н. Программирование на C# для начинающих. Особенности языка / А. Н. Васильев. – Москва : Эксмо, 2019. – 528 с.

2. Гриффитс, И. Программируем на C# 8.0. Разработка приложений / И. Гриффитс. – СПб : Питер, 2021. – 944 с.

3. Прайс, М. C# 10 и .NET 6. Современная кросс-платформенная разработка / М. Прайс. – СПб. : Питер, 2023. – 848 с.

4. Танвар, Ш. Параллельное программирование на C# и .NET Core / Ш. Танвар, пер. с англ. А. Д. Ворониной; ред. В. Н. Черников. – М. : ДМК Пресс, 2021. – 272 с.: ил.

5. Троелсен, Э. Язык программирования C# 7 и платформы .NET и .NET Core 8-е изд. / Э. Троелсен, Ф. Джепикс. : Пер. с англ. – СПб. : ООО “Диалектика”, 2018. – 1328 с.

Учебное издание

Швыров Вячеслав Владимирович
Капустин Денис Алексеевич
Шишлакова Виктория Николаевна

ПРОГРАММИРОВАНИЕ ДЛЯ ПЛАТФОРМЫ .NET
Часть 3. РАБОТА С БАЗАМИ ДАННЫХ И МНОГОЗАДАЧНОЕ ПРОГРАММИРОВАНИЕ

Лабораторный практикум

В авторской редакции
Редактор – Швыров В. В.
Дизайн обложки – Швыров В. В.
Корректор – Шишлакова В. Н.
Верстка – Швыров В. В.

Подписано в печать 06.06.2024. Бумага офсетная.
Гарнитура Times New Roman.
Печать ризографическая. Формат 60×84/16. Усл. печ. л. 10,7.
Тираж 50 экз. Заказ № 47.

Издательство ЛГПУ
ФГБОУ ВО «ЛГПУ»
ул. Оборонная, 2, г. Луганск, ЛНР, 291011. Т/ф: +7-857-258-03-20
e-mail: knitaizd@mail.ru