ЯЗЫК ПРОГРАММИРОВАНИЯ C# И ТЕХНОЛОГИЯ .NET

Автор	Иманбердиев Абулхаир Ерболович, Магистр, Главный архитектор-разработчик АО ГТС
Рецензент	Рогозин Олег Викторович, к.т.н., доцент МГТУ им. Н.Э. Баумана

Казахстан, 2022 г.

Содержание

ПРЕДИСЛОВИЕ

В данном методическом пособии читателю раскрываются вопросы по языку C# и технологии .NET. C# - это язык программирования высокого уровня, который используется для создания веб-приложений, сетевых инструментов, 3D-приложений, утилиты для автоматизации. Также этот язык часто используется специалистами в сфере информационной безопасности.

В данном пособии будут рассмотрены следующие темы:

- Основные инструкций и операторы языка C#.
- Объектно-ориентированное программирование.
- Работа с данными из баз данных, технология LINQ.
- Работа с файлами и потоками.

Тщательно изучив данное методическое пособие вы получите все необходимые знания для того, чтобы начать работать *.Net*-программистом.

ГЛАВА 1. МЕТОДЫ

Любое приложение в C# начинает работать с метода **Main()**.

Статические методы – методы классов, которых можно вызывать без создания каких-то объектов.

Если в классах объявлены методы с модификатором **static** то, это значит, что подобные методы можно вызывать без создания объекта класса, с помощью **точки доступа:**

```
имя_класса.имя_статического_метода();
```

Вызов метода:

1) **По имени:** если метод находится в этом же классе.
2) **Получить доступ через имя класса(если метод static):**
   - **имя_класса.имя_метода();**
   - метод должен быть объявлен с модификатором доступа **public**.

Для возвращения значение из метода используется оператор **return**. После оператора **return** пишется выражение, которая должна соответствовать типу **возвращаемого_значения** метода. Если же тип возвращаемого_значения – **void,** то это значит фактически, что он не возвращает ничего. То есть, это просто какой-то метод, который выполнил свою функцию и оператор **return** может в этом случае не быть в методе. Если метод имеет тип возвращаемого значение **НЕ void** , то он в любом случае должен возвращать какое-то значение соответствущего типа.

Если мы хотим где-то выйти из метода, то мы можем написать **return** без параметра. В этом случае, мы можем выйти из метода с возвращаемым_значением **void**. Это называется безусловный выход.

Параметры метода:

1) Указывается тип и имя параметра;
2) Параметры разделяются запятыми;
3) Допустимо значение по умолчанию. При вызове “метода с параметрами со значениями по умолчанию” в параметры со значением по умолчанию мы можем не передавать значение параметра, то есть в таком случае в этот параметр будет передан значение по умолчанию. Иными словами, при вызове метода с параметром, у которого есть значение по умолчанию, мы можем не передавать значение, тогда при вызове метода значением такого параметра устанавливается значение по умолчанию. А в параметры метода без значения по умолчанию, мы должны передавать значения при вызове этого метода. Пример:

```
static double Add(double x, double y = 7)
{
        return x+y;
}
```

где y – параметр со значением по умолчанию

Все “параметры со значением по умолчанию” должны идти в конце, то есть после параметров без значения по умолчанию. Также, при вызове метода мы можем также

передавать значение к “параметрам со значением по умолчанию”, в таких случаях, мы переписываем на новые значения “значения параметров по умолчанию”.

Если у нас есть много параметров со значением по умолчанию и со значениями без умолчания, и если мы хотим задать (перезаписать) значение только некоторых параметров, то мы должны явно указывать **имя_параметра** и **значение**. Синтаксис:

имя_метода(имя_парам1: знач1, имя_парам2: знач2);

Пример:

Add(x:56, z:89);

Но здесь нам дается выбор перезаписывать или нет только с параметрами со значениями по умолчанию, а параметры со значениями без умолчания мы должны задать обязательно.

Есть также возможность создавать методы с переменным списком параметров. Этот подход удобен, когда мы хотим создать универсальный метод, который будет принимать большое количество параметров. Здесь мы можем использовать массив в качестве параметра. Но при передаче значение параметром при вызове этого метода мы создаем объект типа массив и передаем значение через этот объект. Пример:

Пример 1

```
using System;
    class Program
    {
        static void Main(string[] args)
        {
            double z = Add(new double[]{5, 6, 7});
            Console.WriteLine(z);
            Console.Read();
        }

        static double Add(double[] a)
        {
            double temp = 0;
            foreach (double s in a)
            {
                temp += s;
            }
            return temp;
        }
    }
```

Результат: 18

Существует способ описать механизм передачи переменного списка параметра. Чтобы определить метод, принимающий переменное количество параметров нам необходимо дописать ключевое слово **“params”** перед объявлением массива. В таком случае, мы можем передавать значения параметрам через запятую, то есть без создания экземпляра массива. Пример:

Пример 2

```
using System;

    class Program
    {
        static void Main(string[] args)
        {
            double z = Add(5, 6, 7);
            Console.WriteLine(z);
            Console.Read();
        }

        static double Add(params double[] a)
        {
            double temp = 0;
            foreach (double s in a)
            {
                temp += s;
            }
            return temp;
        }
    }
```

Результат: 18

Способы передачи параметров:

in	по значению
ref	по ссылке
out	возвращаемое значение

1) **При передаче по значению** происходит копирование значений, которые мы передаем на метод при его вызове, на метод. Потом метод будет обрабатывать копию, а на оригинал никак не воздействует.

При создании объекта класса мы передаем переменной ссылку на класс:

```
имя_класса имя_переменной = new имя_класса();
```

Переменная содержит всего лишь адрес объекта класса. Это означает, что когда мы передаем по значению мы создаем копию переменной, который будет содержать адрес объекта класса. Но это будет один и тот же объект, то есть здесь будет копия ссылки, которая будет содержать адрес того же самого объекта.

2) **Передача параметров по ссылке**. При передаче параметра по значению, когда мы меняем значения переменных в методах, где они поступают как параметры мы меняем всего лишь их копию. А в случае **передачи по ссылке,** во-первых, для того чтобы передать параметры по ссылке мы должны при определении параметра добавить ключевое слово **ref**. **ref** будет означать, что мы передаем параметры по ссылке, то есть фактически мы не создаем копию переменных, а мы создаем отдельную переменную, которая будет содержать адрес нашей переменной. И через нее мы будем модифицировать нашу исходную переменную. При вызове метода нам тоже надо написать ключевое слово **ref** перед тем параметром, которая передается по ссылке. Почему это делается? – Потому что, на самом деле могут быть два разных метода. Один метод может принимать наши

параметры по значению, другой может по ссылке. Это будет два разных метода. **При передаче параметров по ссылке мы работаем не с копией переменной, а работаем с ссылкой на наш объект.** То есть передается не копия, а передается некий объект, который является адресом нашей переменной. И через этот адрес мы работаем с исходным значением. Пример способу передачи параметров по ссылке:

Пример 3

```
using System;

    class Program
    {
        static void Main(string[] args)
        {
            double x = 56;
            double y = 67;
            double z = Add(ref x,y);
            Console.WriteLine(x);
            Console.ReadLine();
        }

        static double Add(ref double x, double y)
        {
            x++;
            return x + y;
        }
    }
```

Результат: 57

3) **Передача параметров с помощью возвращаемого значение**. Этот метод используется, когда мы не хотим возвращать значение с помощью метода (то есть используя оператор **return**). Тогда мы можем используя этот "**метод передачи параметров с возвращающим значением**" вернуть возвращаемое_значение в одном из параметров. Для этого мы перед параметром пишем ключевое слово **out**. Это означает, что этот параметр будет использоваться только для того чтобы присвоить ему какие-то значение внутри метода.

Он не может использоваться до инициализации в методе, и главное при вызове этого метода мы не инициализируем этот параметр, потому что компилятор видит, что он будет инициализирован внутри метода. И при вызове этого метода мы перед таким параметром (**out парам3**) также должны указывать ключевое слово **out**. Пример:

```
имя_метода(парам1, парам2, out парам3);
```

В принципе, **out**-параметров в сигнатуре может быть сколько угодно, это удобно когда у нас несколько возвращаемых значений, и когда с помощью **return** не справиться, потому что **return** всегда возвращает только что-то одно. Пример:

Пример 4

```
using System;

class Program
{
```

```
    static void Main(string[] args)
    {
        double x = 56;
        double y = 67;
        double z;
        Add(x, y,out z);
        Console.WriteLine(z);
        Console.ReadLine();
    }

    static void Add(double x, double y, out double z)
    {
        z = x + y;
    }
}
```

Результат: 123

СИГНАТУРА МЕТОДОВ

Компилятор различает методы по сигнатуре.

Сигнатура метода – набор параметров, по которым можно однозначно идентифицировать метода.

В сигнатуру входят: **имя метода**, то есть по имени метода компилятор может различить методы, но может существовать несколько методов с одинаковым именем. Поэтому компилятор также разделяет методы по типу и количеству параметров, и по порядку параметров. А имена параметров к сигнатуре не относятся. Возвращаемое значение метода в сигнатуру не входит. В сигнатуру также входят модификаторы, позволяющие выполнять передачу по ссылке (**ref**) или реализовать параметр как возвращаемое значение(**out**). Именно поэтому при вызове соответствующего метода мы должны указывать ключевые слова **ref** и **out**, чтобы дать компилятору понять какой именно метод вызвать. Заключая повторим, **в сигнатуру входят: имя_метода, типы параметров, их количество, их порядок и модификаторы, позволяющее задать различные варианты передачи параметров**. А имена параметров и возвращаемое значение в сигнатуру не входят.

ГЛАВА 2. ОБРАБОТКА ОШИБОК

Исключения позволяют непросто вернуть какое-то значение, которое можно интерпретировать абсолютно по-разному, а вернуть сразу целый объект. Причем объект, который построен по определенным правилам, объект несет в себе всю необходимую информацию об исключений. То есть, это – непросто код ошибки, это – текст связанный с ошибкой, это – какая-то информация из стека приложений, который содержит вспомогательные данные, позволяющий понять в чем суть ошибки. Это – тип самого объекта, который также позволяет легко определить что произошло в программе.

Использование блока try…catch:

1) **код, генерирующий ошибку, размещают в блок try.**

2) **блок catch позволяет описать тип объекта, содержащего информацию об ошибке и создать переменную.**

Пример:

```
try {
      Console.WriteLine("Enter a number");
      int i = int.Parse(Console.ReadLine());
}
catch (OverflowException caught)
{
      Console.WriteLine(caught);
}
```

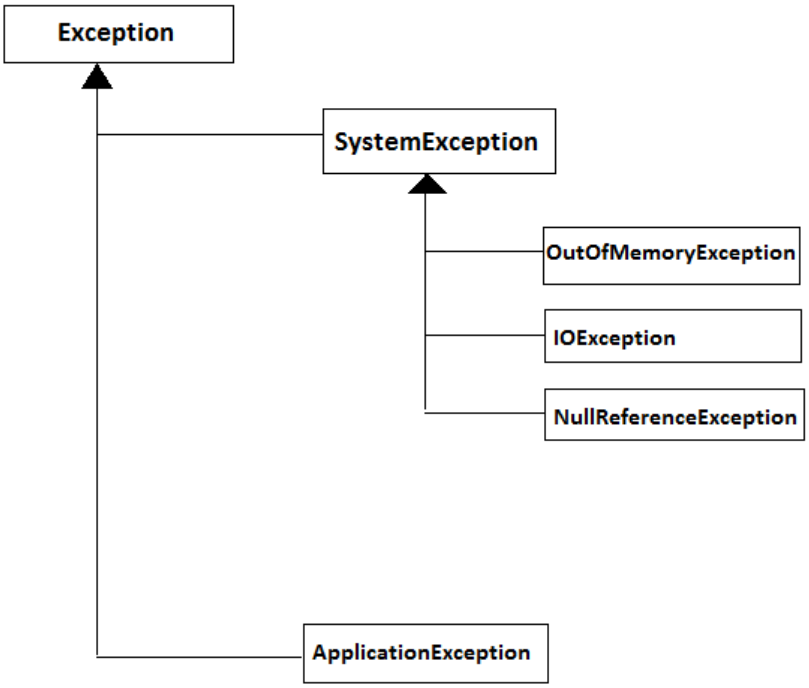
В этом примере показывается как перехватить исключение, связанная с переполнением. Переменная **int** может иметь значение в интервале **-32 тыс. - +32 тыс.** В случае если мы попытаемся ввести значение больше чем допустимый размер, то выполняется специальное исключение **OverflowException**. **OverflowException –** тип, на основе которого создается объект. Именно, внутри этого объекта храниться вся необходимая информация об ошибке, то есть если передаваемое в **int i** наше значение больше допустимого значение вызывается код, который внутри блока **catch**.

Потенциально весь код, который может генерировать исключение размещается внутри блока **try{…}.**

try – ключевое слово языка программирования C#. И оно фактически говорит “попробуй выполнится” и если все нормально, то мы переходим к выполнению операторов, которые следует после блоков **try** и **catch**. То есть, блок **try** выполняется и он способен сгенерировать какое-то исключение. И если же действительно произошло какая-та ошибка, то выполняется блок **catch**.

Обработка ошибок осуществляется внутри блоков **try-catch**. Внутри блока **try** записывается код, который способен генерировать ошибки, причем, этот код может генерировать любое количество ошибок. И после блока **catch** может перехватывать эти ошибки. Ну поскольку код внутри блока **try** большой, то логично, что блоков **catch** может быть последовательно несколько. И все они могут перехватывать разные исключение.

Иерархия исключений:



Все определенные исключения наследуются от класса **Exception**. И мы можем спокойно его использовать.

SystemException – набор системных исключений. И все классы, которые наследуются от класса **SystemException** считаются типами исключение, который так или иначе генерирует **.Net Framework**. Это означает, что если вам попадает что-то связанное с **SystemException** вероятнее всего у вас произошла проблема при работе одного из методов **.Net Framework**, либо же вы пытаетесь выделить больше памяти чем у вас есть(**OutOfMemoryException**), либо же у вас есть какие-то проблемы с вводом-выводом(**IOException**). Все системные исключение наследуются от **SystemException**.

Также есть набор исключений, которые наследуются от класса **ApplicationException**. Это – ошибки, связанные непосредственно с выполнением программы.

Если вы хотите создать собственное исключение – свой собственный класс, который описывает вашу ошибочную ситуацию, то вам надо наследоваться от класса **Exception**. Это – важно.

НЕСКОЛЬКО БЛОКОВ CATCH

1) Если у вас код, который может потенциально генерировать несколько исключений(код в блоке try может быть сколь угодно большим, хоть всю программу туда запишите), то нам необходимо описать целый ряд блоков **catch**, которые позволяют перехватывать различные исключения. Пример (попытка перехватить сразу 2 исключение):

```
try
{
        Console.WriteLine("Enter first number");
        int i = int.Parse(Console.ReadLine());
        Console.WriteLine("Enter second number");
        int j = int.Parse(Console.ReadLine());
        int k = i / j;
}
catch (OverflowException caught) {…}
catch (DivideByZeroException caught) {…}
```

Если данные пользователем введены правильно, то блок try выполниться полностью и блоки catch будут игнорироваться. А если внутри блока try произошло исключение, в нашем примере деление на ноль или же произошло переполнение, то в зависимости от типа ошибки срабатывает тот или иной блок catch. Внутри блока catch, естественно, мы можем аварийно завершить программу или же можем продолжить программу. То есть, что записывать внутри блока catch решать нам, но необходимо очень хорошо помнить о том, что все исключения объединены в некую иерархию. Если вдруг у нас есть целая иерархия исключений, и если вдруг мы действительно будем обрабатывать сразу много исключений, то необходимо понимать что сначала необходимо обрабатывать исключения, которые базируется на объектах от классов производных, то есть, которые находятся в самом низу иерархий. А затем уже мы начинаем обрабатывать исключения, которые базируется на базовых классов. В частности, самым последним, которые мы должны обрабатывать это исключение, базирующее на классе **Exception**, потому что блок catch, обрабатывающий исключения родительского типа, он всегда будет перехватывать и все дочерние исключения. То есть, здесь необходимо об этом правиле помнить и в случае если вы хотите, чтобы ваше исключение обрабатывалась в любом случае и программа аварийно не завершилось, то необходимо вспомнить про иерархию исключения, и обрабатывать сначала исключение дочерних типов, а потом уже родительских. Пример:

```
using System;
namespace ExceptionExample
{
    class Program
    {
        static void Main(string[] args)
        {
        try
            {
                int i = F(44); // тут идет переполнение допустимого значения int
                Console.WriteLine(i);
            }
        catch (OverflowException ex) // ex -обьект класса - исключения
OverflowException
            {
                Console.WriteLine("Overflow");
            }
            Console.ReadLine();
        }


        static int F(int n)
        {
            checked
```

```
            {
                if (n == 1) return 1;
                return n * F(n - 1);
            }

        }
    }
}
```

Результат: `Overflow`

У исключении есть иерархия. К примеру, **OverflowException** наследуется от **SystemException**, а сам **SystemException** наследуется от класса **Exception.** Таким образом, если мы попытаемся перехватить исключения базового типа, то блок catch может перехватить не только то, что непосредственно порождено от класса **Exception,** не только те объекты, которые являются объектами класса **Exception,** но и все любые объекты, которые являются объектами класса порожденных, так или иначе от класса **Exception**. То есть, **Exception** ловит все. Пример:

```
using System;
namespace ExceptionExample
{
    class Program
    {
        static void Main(string[] args)
        {
            try
            {
                int i = F(44); // тут идет переполнение допустимого значения int
                Console.WriteLine(i);
            }
            catch (Exception ex) // ex -обьект класса - исключения Exception
            {
                Console.WriteLine("Overflow");
            }
            Console.ReadLine();
        }


        static int F(int n)
        {
            checked
            {
                if (n == 1) return 1;
                return n * F(n - 1);
            }

        }
    }
}
```

Результат: `Overflow`

А происходит так, потому что в ссылки на базовые классы всегда можно присвоить объект производного класса. То есть, в ссылку базового класса **Exception** мы можем присвоить объект производного класса **OverflowException**:

```
Exception ex = new OverflowException();
```

Компилятор берет объект, который вернул ему код в качестве ошибки и начинает последовательно проходится по всем блокам **catch** и смотреть можно ли ссылку на объект сгенерированного типа записать в одно из переменных, которая содержится в блоке **catch**. Поскольку в переменную от базового класса **Exception** можно записать ссылку на объект любого производного типа, то **catch – Exception** будет совершенно спокойно срабатывать. Как только какой-то **catch** сработает и если у нас есть большая последовательность блоков **catch** они уже будут проигнорированы. То есть, типы проверяются до того как мы найдем первый **catch**. И если мы находим первый **catch**, выполняем все что находиться в блоке **catch**, после выполняется последующий код после блоков **catch**, а остальные **catch** игнорируется.

ГЕНЕРАЦИЯ ТИПА ИСКЛЮЧЕНИЯ

Если мы хотим переопределить реакцию на какую-то ошибку, то мы должны создать объект типа **Exception** или объект класса производного от **Exception**. И каким-то образом передать его в вызывающий код. При этом надо аварийно завершить выполнение метода. Для того, чтобы передать объект типа **Exception** в вызывающий код необходимо использовать ключевое слово “**throw**”. Необходимо использовать это слово для того, чтобы вернуть объект типа **Exception** в вызывающий код. Ну а в качестве параметра это ключевое слово получает ссылку на объект(ссылку на объект типа **Exception**). Пример:

```
using System;
namespace ExceptionExample
{
    class Program
    {
        static void Main(string[] args)
        {
            try
            {
                int i = F(-1); // тут идет переполнение допустимого значения int
                Console.WriteLine(i);
            }
            catch (OverflowException ex) // ex -обьект класса - исключения Exception
            {
                Console.WriteLine("Overflow");
            }

            catch (Exception ex)
            {
                Console.WriteLine("Error");
            }
            Console.ReadLine();
        }


        static int F(int n)
        {
            if (n < 1)
            {
              throw new Exception(); // с помощью throw выбрасываем обьект класса
Exception наружу
            }
            checked
            {
                if (n == 1) return 1;
                return n * F(n - 1);
```

```
            }

        }
    }
}
```

Результат: Error

Также мы можем создавать собственные классы исключения. В этом случае, у нас появится возможность создавать объекты таких классов и выбросить их в выполняемый код. Пример:

```
using System;
namespace ExceptionExample
{
    class Program
    {
        static void Main(string[] args)
        {
            try
            {
                int i = F(-1); // тут идет переполнение допустимого значения int
                Console.WriteLine(i);
            }
            catch (OverflowException ex) // ex -обьект класса - исключения Exception
            {
                Console.WriteLine("Overflow");
            }

            catch (InvalidParameterException ex) // ex - обьект класса- исключения
InvalidParameterException
            {
                Console.WriteLine("Invalid parameter");
            }
            catch (Exception ex)
            {
                Console.WriteLine("Error");
            }
            Console.ReadLine();
        }


        static int F(int n)
        {
            if (n < 1)
            {
                throw new InvalidParameterException(); // с помощью throw выбрасываем
обьект класса InvalidParameterException в выполняемый код
            }
            checked
            {
                if (n == 1) return 1;
                return n * F(n - 1);
            }
        }
    }


    class InvalidParameterException : Exception
    {

    }
```

```
}
```

Результат: Invalid parameter

Кроме **catch(Exception ex){…}** можно перехватывать все исключения с помощью вот такой конструкций: **catch{…}**. **catch{…}** перехватывает все исключения, как и **catch(Exception ex){…}**.

В случае, если блок **catch** не должен гасить исключение, он просто должен каким-то образом нотифицировать пользователя, а затем пробросить исключения дальше, то это можно сделать используя оператор **throw** без параметра: **throw;** в блоке **catch**. Это позволяет пробросить текущее исключение дальше сохранив всю информацию из стека, сохранив все данные, которые были.

Существует еще один блок – блок **finally{…};** . Он позволяет выполнить код независимо от того было исключение или не было его. Пример:

```
using System;
namespace ExceptionExample
{
    class Program
    {
        static void Main(string[] args)
        {
            try
            {
                int i = F(-1); // тут идет переполнение допустимого значения int
                Console.WriteLine(i);
            }
            catch (OverflowException ex) // ex -обьект класса - исключения Exception
            {
                Console.WriteLine("Overflow");
            }

            catch (InvalidParameterException ex) // ex - обьект класса- исключения
InvalidCastException
            {
                Console.WriteLine("Invalid parameter");

            }
            catch (Exception ex)
            {
                Console.WriteLine("Error");
            }
            finally
            {
                Console.WriteLine("Finally");
            }
            Console.ReadLine();
        }


        static int F(int n)
        {
            if (n < 1)
            {
                throw new InvalidParameterException(); // с помощью throw выбрасываем
обьект класса InvalidParameterException в выполняемый код
            }
            checked
            {
                if (n == 1) return 1;
                return n * F(n - 1);
```

```
            }
        }
    }

    class InvalidParameterException : Exception
    {

    }
}
```

Результат: Invalid parameter
Finally

ГЛАВА 3. ОБЪЕКТНО-ОРИЕНТИРОВАННОЕ ПРОГРАММИРОВАНИЕ В C#

Понятие объектно-ориентированного программирования:

1) **Класс** – абстракция, описание чего-либо.

2) **Объект** – экземпляр класса.

Класс – это инструкция, которая описывает как нам создавать объекты, то есть это – описание чего-либо. Иными словами, первое что мы делаем в C# создавая любое приложение, это описываем наши классы. То есть, мы описываем инструкций, которые позволят нам в дальнейшем во время работы приложения создавать объекты этих классов.

Основное отличие между структурами и классами состоит в том, что структуры больше привязаны к стеку, которая отдельно выделяется для приложения, где создаются объекты простейших типов. А классы, естественно, механизм их создания привязан к общей памяти, которая выделяются приложению, и собственно говоря, в глобальной области памяти они создаются. Наличие структуры в языке программирования – это не признак ООП. Это – всего на всего признак поддержки абстрактных типов данных. А основным инструментом ООП является **класс**.

Инкапсуляция – это объединение данных и методов по работе с этими данными внутри класса. То есть, инкапсуляция подразумевает то, что когда мы собираемся создавать инструкцию для создании какого-то объекта, то мы внутри класса описываем данные и внутри класса мы описываем методы по работе с этими данными. В структурах тоже можно описывать данные и методы по работе с этими данными, но внутри классов к этому всему добавляется еще такое понятие как модификатор доступа. Когда мы говорим про ООП мы не говорим про методы статические, мы говорим уже про методы класса. Эти методы объявляются без модификатора **static**, их можно вызвать только через имя объекта этого класса.

Каким образом в C# мы разграничиваем доступ к тем или иным данным, тем или иным методам? – Мы используем модификаторы доступа. Модификаторы доступа в C#:

1) **public –** открытый доступ. Это значит, что мы можем создав объект класса вызвать этот метод без всяких ограничений. Модификатор **public** определяет, что данные или методы в конкретном объекте будут доступны всем. Вы можете просто получить имя объекта, и через имя получать доступ действительно ко всем данным и методам без каких-либо ограничений. Именно, по этой причине обычно данные с модификатором **public** не определяются.

2) **private** – закрытый доступ. Это значит, что элементы с модификатором **private** доступны только внутри этого класса.

3) **protected** говорит о том, что снаружи эти методы недоступны, но наследнику доступны.

СОЗДАНИЕ ОБЪЕКТА

Все объекты класса создаются с помощью оператора **new**. Как только мы указываем **new**, после этого мы указываем тип, который мы создали, то есть описание. И тут мы получаем фактически адрес созданного объекта в общей памяти. То есть, оператор **new** выделяет память под наш объект и возвращает на него ссылку. Создание объекта типа структуры Time(пример):

```
Time имя_перем;
```

Когда мы говорим про классы, то здесь надо обязательно использовать оператор **new**. Единственный способ создания объекта класса в C# только с помощью оператора **new**.

С помощью оператора **new** создается объект какого-то типа, и возвращается адрес этого объекта. В результате в переменную типа того класса записывается адрес этого объекта. И вот сама переменная-ссылочная переменная, к ней работает все те принципы, которые работают по отношению к ссылочным переменным. То есть, этот адрес можно присвоить в переменную аналогичного типа. Соответственно, у вас будет два переменных с одинаковым адресом, указывающих на один и тот же объект. Пример:

имя_класса имя_перем = new имя_класса();

где **имя_перем** содержит адрес объекта.

Для того, чтобы вызвать какие-то методы или получить доступ к данным мы используем тот же самый синтаксис, который использовали и для вызова статических методов, или доступа к каким-то статическим данным:

имя_перем.имя_метода();

имя_перем.имя_поля;

КЛЮЧЕВОЕ СЛОВО “**this**”

Внутри класса всегда доступна специальная ссылка на себя. Эта ссылка “**this**”. Как только мы создаем объект класса внутри класса всегда существует ссылка, которая содержит адрес созданного нами объекта. Мы этим можем пользоваться для того, чтобы инициализировать поля объекта. К примеру, мы можем инициализировать значение “поле” значением параметра, когда у них одинаковые имена. Причем, мы тут указывая поле с помощью ключевого слово **this** обращаемся к его ссылке и тем самым можем различать поле от параметра:

```
class BankAccount
{
    ...
    public void SetName(string name)
    {
        this.name = name;
    }
    private string name;
}
```

Очень часто внутри класса могут быть методы, которые принимают в качестве параметров объекты этого же класса. Если мы вдруг захотим вызвать этот метод передав в качестве параметров себя, то **this** мы можем использовать в этих целях. Внутри класса всегда есть ссылка на созданный объект. И “**this**”, как только мы создаем объект с помощью

оператора **new**, **this** всегда инициализируется в адрес этого объекта. Так что внутри класса мы сможем его использовать для доступа к методам, и для доступа к данным. Несмотря на то, что это - ссылка на объект, эта ссылка находится внутри и компилятор в общем-то понимает что мы с помощью нее можем обращаться не только к **public**, но и ко всей структуре нашего класса(приватным данным).

СОЗДАНИЕ ВЛОЖЕННЫХ КЛАССОВ

```
class Program
{
    static void Main( )
    {
        Bank.Account yours = new Bank.Account( );
    }
}
class Bank
{
    ... class Account { ... }
}
```

Классы могут быть вложены. Также как и объявление данных, методов по работе с этими данными внутри класса мы можем объявлять другие классы.

Когда мы создаем вложенный класс и хотим создать объект этого вложенного класса мы используем синтаксис:

имя_внешнего_класса.имя_внутренного_класса имя_перем = new имя_внешнего_класса. имя_внутренного_класса();

Причем, тип вложенного класса должен быть **public**. Ну а если у нашего вложенного класса **модификатор_доступа** будет **private**, мы естественно не сможем создавать объекты этого класса.

НАСЛЕДОВАНИЕ

Базовый класс удобно использовать для реализаций общей идеи, а в дочерних классах мы прописываем логику индивидуально для каждого типа используя логику базового класса. Базовый класс будет обладать свойствами какими-то общими, и их можно в последующем использовать. Все классы в C# являются наследниками от **object**.

Object – самый базовый класс. Например, внутри него есть замечательный метод **ToString()**, и этот метод, естественно, есть у любого объекта – абсолютно у любого класса в C#. Наследование позволяет создавать новое описание, включающее все данные и методы уже существующего описания. Независимо от того какие данные объявлены в базовом классе(**private**, **public**), они в любом случае переходят в производный класс. Даже если они были объявлены как **private**, они никуда не исчезают из производных классах, к ним просто нету доступа, но они есть. Если у нас есть метод с модификатором **protected**, который ты

можешь вызвать и работать с **private** данными, то в производном классе ты его тоже можешь вызвать и он будет работать с теми же данными. Хотя напрямую с класса мы к ним уже доступ не получим.

При наследований все данные и методы – они переходят в производный класс. В производном классе мы сможем добавить собственные данные, методы, расширить функциональность класса. Но при этом объект производного класса будет всегда включать данные и методы базового класса.

По причине того, что производные классы включают все данные и методы уже существующего базового класса, в C# отсутствует множественное наследование, хотя в C++ оно есть.

ПОЛИМОРФИЗМ

Полиморфизм позволяет использовать ссылки на базовые типы, поддерживая реализацию, определяемую реальным объектом.

Полиморфизм – это возможность класса наследника менять реализацию родительского класса сохраняя его интерфейс(**virtual** и **override**). Мы можем переопределять реализацию методов базовых классов в дочерних классах.

Синтаксис создание **наследование**:

```
class имя_производного_класса: имя_базового_класса
{
…
}
```

В ссылки на базовый класс мы можем присвоить объект производного класса. Это работает, потому что в производном классе всегда есть методы, которые были описаны в базовом классе.

ГЛАВА 4. ССЫЛКИ

Объект класса – экземпляр класса, который содержит данные и методы по работе с этими данными. И естественно, чтобы работать с выделенной областью памяти нам необходимо какой-то механизм, то есть нам необходимо какие-то переменные через которые мы сможем получать доступ к данным объекта, к методам объекта, которые определены внутри объекта и которые позволяют работать с этими данными. Вот этот механизм является **ссылкой**. То есть, **ссылка –** простая переменная, задача которой с одной стороны содержать адрес объекта в памяти, то есть ссылка знает с каким объектом мы работаем, и естественно, с помощью ссылки мы имеем возможность каким-то образом влиять на этот объект, мы имеем возможность получать доступ к данным, которые определены как открытые, мы имеем возможность получать доступ к публичным методам, которые позволяют работать с данными. То есть, ссылка – некая переменная(заданного типа, то есть когда мы создаем мы должны указывать тип ссылки), которая содержит адрес объекта. Несколько ссылок могут указывать на один и тот же объект.

.Net Framework проверяет объекты на наличие каких-то ссылок. Если ни одной ссылки на объект нету, .Net Framework, а именно “сборщик мусора” принимает решение объект надо уничтожить, то есть объект больше не используется в приложении. Если в C++ нам надо было явно следить за тем чтобы уничтожить объект в нужном месте в нужное время, то в C# этого не нужно делать. Мы можем явно сказать, что ссылка на объект не указывает. Мы можем сказать через ключевое слово “**null**” сказать, что ссылка никуда не указывает, то есть ссылка указывает на “**null**” и мы фактически своеобразным образом ссылку очищаем. Когда мы создаем ссылку по умолчанию оно инициализируется значением **null**. То есть, создание самой ссылки еще не повод сразу же получить доступ к каким-то данным, методам, а ссылка должна получить адрес какого-то объекта. Адрес объекта возвращает оператор **new**. А с помощью **null** мы можем обнулить ссылку.

Существует 3 вида передачи параметров:

1) С помощью ключевого слово “**ref**”. Это – передача параметров по ссылке.
2) Возвращаемое значение “**out**”.
3) Передача параметров по значению.

Несмотря на то, что если мы создаем метод, который принимает в качестве параметра ссылку на объект какого-то класса, это означает, что здесь создается копия нашей первоначальной ссылки, но при этом несмотря ни на что, поскольку передача по ссылке – это – адрес объекта.

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace Modul8
{
    class Program
    {
        static void Main(string[] args)
        {
            MyClass myclass = new MyClass();
            myclass.data = 10;
            IncreaseData(myclass);
            Console.WriteLine(myclass.data);
            Console.ReadLine();
        }

        // Метод, который принимает ссылку на класс(MyClass)
        static void IncreaseData(MyClass c)
```

```
        {
            c.data++;
        }}

    class MyClass
    {
        public int data;
    }
}
```

Результат: 11

```
using System;
namespace Modul8
{
    class Program
    {
        static void Main(string[] args)
        {
            MyClass myclass = new MyClass();
            myclass.data = 10;
            IncreaseData(ref myclass);
            Console.WriteLine(myclass.data);
            Console.ReadLine();
        }

        // Метод, который принимает ссылку на класс(MyClass)
        static void IncreaseData(ref MyClass c)
        {
            c = new MyClass();
            c.data++;
        }}

    class MyClass
    {
        public int data;
    }
}
```

Результат: 1

Если мы хотим использовать ссылки в качестве возвращаемого значения, то используется **out**, а если мы действительно хотим получить параметры, то **ref**.

В C# все классы наследуются от класса **object**. Несмотря на то, что мы явно не определяем наследование от **object**, даже в таких случаях все классы наследуются от **object**.

В дочерних классах нам будет доступно всё(методы, поля), что объявлено в родительских классах как **public**, **protected**. А если учитывать то, что все классы неявно наследуются от **object**, то во всех классах доступны члены(методы, поля) класса **object**.

В ссылки на базовый класс можно присвоить объекты производного класса. И с помощью этой ссылки мы можем получать доступ к методам базового класса и дочернего класса. Это возможно потому, что все члены базового класса доступны и в дочерних классах, и тем самым, это позволяет нам вызывать члены и базового, и производного класса:

```
имя_базового_класса имя_перем. = new имя_производного_класса();
```

Если мы хотим преобразовать ссылку типа базового класса к типу производного класса, то мы используем **явное приведение**:

```
((тип_на_который_нужно_преобразовать)имя_ссылки);
```

ОПЕРАТОР **“is”** И **“as”**

Оператор **“is”** позволяет проверить можно ли ссылку(переменную) какого-то типа преобразовать в другой тип(возвращает значение типа **bool**):

```
if(имя_ссылки is целевой_тип){…}
```

То есть, если мы сможем привести ссылку в целевой тип, возвращается **true**. Если “нет”, то **false**.

С помощью оператора “as” мы можем преобразовывать ссылку с одного типа на вторую. Работает эквивалентно “явному приведению”:

```
имя_ссылки as имя_целевого_класса;
```

Если оператор **“is”** возвращает булевое значение, то оператор **“as”** возвращает ссылку на объект, тем самым позволяет привести ссылку к целевому типу.

Различие между **“явным привидением”** и **“оператором as”** в том, что:

1) Если мы используем явное приведение, и явное приведение не сработало, то там будет исключение.

2) Если мы используем оператор **as**, и если оператор **as** не сработало, то будет возвращено **null**.

Ссылки указывают на объекты где-то в памяти. Переменные простых типов – структур(переменные значения) создаются в стеке. В C# определив переменную простого типа, мы можем присвоить его в переменные объекта класса. Пример:

```
int i = 5; // переменная простого типа
object a = i;
```

В C# для того, чтобы мы могли передавать переменной значение обычной ссылки используется специальный механизм: **“Boxing”**, **“Unboxing”**.

Всякий раз, когда мы ссылке присваиваем переменную значение, вставляется специальный код, который создает ячейку в памяти. В эту ячейку записывает копию переменной, и наша ссылка будет содержать адрес ячейки памяти, в которой хранится копия переменной. То есть, создается фактически своеобразный маленький объект. Затем как только мы пытаемся использовать эту переменную, а чтобы его использовать нам надо преобразовать явно ссылку к типу значения. Пример:

```
int i = 5; // переменная значение
```

```
object a = i; // a – ссылка
(int)a; // распаковка(Unboxing)
```

Метод может принимать параметры:

1) **по значению** (тут производится копирование данных, и мы работаем с копиями, и любые изменения не затронут исходный объект).

2) **передача по ссылке** предполагает, что мы фактически упаковываем нашу переменную, содержащую либо значение, либо ссылку в специальный объект и затем в методе работаем непосредственно с нашей исходной ссылкой и с нашим исходным значением, то есть здесь **“упаковка”**, затем **“распаковка”** данных.

3) **передача возвращаемых значений**: когда мы передаем ссылку подразумевается, что мы не будем ее инициализировать. Мы подразумеваем, что эта ссылка будет инициализирована значением, который вернет метод в **out – параметре**.

ГЛАВА 5. СОЗДАНИЕ И УНИЧТОЖЕНИЕ ОБЪЕКТОВ

Объект класса создается с помощью оператора **new**. Используя **new** мы выделяем память. **Создание объекта(синтаксис):**

```
new имя_класса();
```

Каждый раз когда мы создаем объект, вызывается какой-то метод. И этот метод называется **конструктором**. Суть конструктора состоит в том, чтобы инициализировать все данные внутри объекта.

Любой класс имеет минимум один конструктор, то есть всегда внутри класса существует метод, который занимается инициализацией. В случае, если мы создаем класс, в котором этого конструктора нету, то конструктор генерируется автоматически. Пример:

```
public A(int age)
{
…
}
```

При создании объекта класса, мы инициализируем параметры конструктора в круглых скобках:

```
A a = new A(5);
```

В это примере мы инициализируем параметр **age** 5. Как только мы определяем конструктор сами, то конструктор по умолчанию пропадает.

Синтаксис создание конструктора:

```
модиф_доступа имя_класса(парам)
{
…
}
```

Конструктор не имеет возвращаемого значение. **Имя_конструктора должен соответствовать с именем_класса.**

В классе может быть много конструкторов, которые принимают различное количество, различные типы параметров. И этим они между собой различаются.

Если у нас в классе есть несколько конструкторов, то иногда нам понадобится вызвать один конструктор из другого конструктора. Это можно сделать используя оператор **this**. Пример:

```
class Date
```

```
{
    public Date(): this(1970, 1, 1){…}
    public Date(int year, int month, int day){…}
}
```

В этом примере мы из конструктора без параметра вызываем конструктор с параметрами, передавая туда значение по умолчанию. Этот метод используется когда мы не хотим дублировать код, а просто ссылаться уже на существующий. То есть, **this** позволяет нам вызвать конструктор этого же класса перед тем, как будет выполнен код конструктора, который использует подобный вызов.

Конструктор – метод, который мы используем при создании объекта.

Конструкторы также могут быть с **модификатором_доступа private**. Идея использование конструктора с **модификатором_доступа private** состоит в том, чтобы контролировать количество объектов нашего класса, который мы создаем. И естественно, чтобы мы могли контролировать количество объектов, мы должны запретить их создание откуда-то снаружи. Поэтому первое, что мы делаем – это объявляем наш класс с конструктором, имеющий **модификатор_доступа private**. То есть, здесь мы запретили создание объектов. Теперь вопрос: “А как же их создавать?” – Здесь действует очень простой способ создания этих объектов: использование статического метода.

В классе мы объявляем статический метод. И этот метод будет иметь доступ ко всем внутренним механизмам класса (потому что, он определяется внутри класса), в том числе и приватному конструктору. Поэтому внутри этого статического метода создаем объект нашего класса. А для того, чтобы контролировать количество объектов ссылку на объект надо вынести в наружу и сделать его статическим(столько статических ссылок, столько и объектов), чтобы ее можно было использовать в статическом методе. Дальше мы создаем объект нашего класса и присваиваем его в статическую ссылку. И после всего этого мы возвращаем нашу ссылку(типа нашего класса). И теперь, когда мы захотим создать объект этого класса, мы вызываем наш статический метод через имя класса:

```
имя_класса имя_ссылки = имя_класса.имя_статического_метода();
```

```
using System;
namespace ConsoleApplication3
{
    class Program
    {
        static void Main(string[] args)
        {
            A a = A.CreateA();
        }
    }

    class A
    {
        int age;

        public A(int age)
        {
            this.age = age;
        }

        static A a;
        public static A CreateA()
        {
```

```
            if (a == null)
            {
                a = new A(10);
            }
            return a;
        }
    }
}
```

И еще один пример использования приватного конструктора: когда класс несет только статические методы(к примеру, класс Console и т.д.), то внутри можно объявить приватный конструктор, чтобы явно дать разработчику понять, что объект этого класса создавать нельзя.

СТАТИЧЕСКИЕ КОНСТРУКТОРЫ

Когда у нас есть статические переменные, и мы хотим их инициализировать, причем динамически, для этого мы используем статический конструктор, который позволяет выполнить инициализацию статических данных. В таких конструкторах нет никаких параметров, потому что этот конструктор вызывается до первого создания объекта класса(до первого использования класса). Синтаксис:

```
static имя_класса()
{
…
}
```

Статические конструкторы:

1) Вызывается в тот момент, когда класс был загружен в память.
2) Используется для инициализации статических переменных.
3) Гарантированно вызывается перед первым вызовом обычного конструктора.
4) Не может принимать параметров и не имеет модификатора доступа.

ЖИЗНЕННЫЙ ЦИКЛ ОБЪЕКТОВ

1) Объекты находятся в памяти, пока существует хотя бы одна ссылка на объект.

2) Объекты уничтожаются с помощью специального механизма – **Garbage Collector(GC)**.

При работе механизма **GC** у старых объектов шансов на выживание значительно больше, чем у молодых. Самым старым объектом во время работы приложения считается **объект типа приложения**.

Принцип работы GC: Он берет все объекты, начинает их просматривать и делает 2 вещи:

1) Он находит объекты, на которых больше нету ссылок, и эти объекты он уничтожает. Уничтожая, он освобождает память. А поскольку блок памяти, выделенный под приложение он в общем-то выделяется одним куском, **GC** еще занимается дефрагментацией, то есть он сдвигает все остальные объекты таким образом, чтобы выделенная память была собрана опять же в виде одного куска. Объект, который не должен быть уничтожен, то есть на него есть хотя бы одна ссылка **GC** помечает специальным номером, этот номер поколений. **Поколение в .Net Framework 3**: нулевое, первое и второе поколение. Всякий раз, когда **GC** пробегает по очереди, он помечает объекты, которые нельзя уничтожать, на которых есть хотя бы одна ссылка поколением +1. То есть, объект

может быть молодым, то есть может находится в поколении ноль, как только его надо уничтожить **GC** его уничтожает, ну а если вдруг найдется одна ссылка на него, то этот объект переходит в поколение один. То есть, задача **GC**:

1) Разбить все созданные объекты на 3 разных поколения для ускорения работы. “Почему для ускорения работы?” – Понимая, что старые объекты – долгожители, **GC** вызывается часто, но к старым объектам практически никогда не обращается. Первая его задача: обработать объекты поколения ноль, то есть самых молодых. И только в том случае, когда объектов поколения ноль становится много, он переходит в поколение один, там он уничтожает тоже лишние объекты. А в случае, если объектов внутри поколения один много, он переходит в поколение два.

Идея **GC** состоит в том, что: 1) Разбить объекты на несколько групп, из которых работать в первую очередь с молодыми. 2) Работать с памятью, как с одной стороны с набором уже созданных объектов, которые занимают непрерывное область памяти. С другой стороны, если все созданные объекты занимают непрерывное область памяти, то оставшиеся место уже не содержит никаких объектов и выделение объектов происходит достаточно быстро. То есть, у нас есть кусок памяти, который занят чем-то и все оставшиеся место. И если нам нужно создать новый объект мы переходим к концу очереди существующих объектов, и выделяем еще один кусок памяти.

GC все время занимается дефрагментацией. За счет дефрагментации и за счет того, что он разбивает все объекты на несколько поколении, и работает в первую очередь с самыми молодыми **GC** работает очень быстро.

Мы можем принудительно вызвать **GC**, для этого существует класс **GC**. У него есть ряд методов, включая методы, которые позволяют принудительно вызвать **GC**. Но контролировать его тяжело. То есть, мы создали объект и мы должны понимать, что мы должны уничтожить его как только на него не будет ни одной ссылки.

Существует также метод **Finalize()**, который вызывается перед уничтожением объекта. **GC** не ждет выполнение метода **Finalize()**. К примеру, у нас есть объект поколения ноль, и **GC** начинает их обрабатывать, он находит объект, который необходимо уничтожить, но он видит, что внутри есть метод **Finalize()**, и тогда он помещает этот объект в специальную очередь и вызывает внутри этой очереди метод **Finalize()**. И переходить к следующему объекту – это означает, что во время прохода **GC** по объектам все объекты, где был метод **Finalize()**, и которые должны быть уничтожены, на самом деле не уничтожаются. Это происходит, потому что надо, чтобы внутри них обработался метод **Finalize()**, поэтому все эти объекты они не уничтожаются, а просто помещаются в очередь финализации. То есть, эти объекты продолжают существовать и у них выполняется метод **Finalize()**, за это отвечает специальный поток финализации, который работает совместно с **GC**. Объект будет уничтожен только через один проход **GC**, когда следующий раз GC пойдет по очереди он увидит, что есть объект, который нужно уничтожить и у него уже обработался метод **Finalize()**, и только после этого этот объект будет уничтожен. Это затормаживает процедуру уничтожения объектов. Если вдруг в поток финализации попадает метод **Finalize()**, который ведет себя неадекватно, то тогда поток финализации вырубится вообще. Это значит, что метод **Finalize()** вообще не будет вызван. Метод **Finalize()** использовать не рекомендуется!

ГЛАВА 6. НАСЛЕДОВАНИЕ И ПОЛИМОРФИЗМ

В C# множественного наследование не существует. В C# существует возможность создание нового класса на основе уже существующего класса(только одного).

Наследование позволяет нам создать новую абстракцию, которая будет в себя включать все данные, все методы предыдущей абстракции. При этом, мы можем легко доопределить собственные данные, мы можем легко доопределить собственные методы, либо же мы можем переопределять методы базового класса, в том случае, если они объявлены с модификатором **protected**, либо **public**. **Синтаксис наследование:**

```
class имя_производного_класса: имя_базового_класса
{
…
}
```

Производный класс содержит все данные и методы, которые были определены в базовом классе. В производном классе мы можем также определить свои данные и методы, или перегрузить методы базового класса.

В производном классе мы легко можем вызывать методы, которые были определены в базовом классе, но в том случае если они были объявлены с модификатором **public** или **protected**.

Модификатор **protected** позволяет скрыть методы базового класса от внешнего воздействия, с другой стороны, он позволяет в производном классе спокойно эти методы использовать. То есть, модификатор **protected** очень часто применяется именно при реализации каких-то шаблонов, связанных с наследованием.

Поскольку, много данных определяются в базовом классе, то где-то надо будет вызывать конструктор, который будет эти данные инициализировать. Особенно, если эти данные объявлены с модификатором **private**. Логично, что в производных классах мы их(данные с модификатором_доступа **private**) инициализировать не можем. Поэтому, в производном классе должен где-то вызываться конструктор базового класса. Если у нас есть конструктор по умолчанию в производном классе или конструктор без параметров, то перед вызовом этого конструктора сначала будет вызван конструктор без параметров базового класса. При этом не важно по умолчанию или нет, главное, чтобы это был конструктор без параметров. Если в производном классе у нас есть конструктор с параметрами, компилятор попробует подставить вызов конструктора без параметров базового класса. В любом случае, если мы ничего явно не указали, компилятор всегда попробует сначала создать объект типа базового класса, он будет считать, что объект типа базового класса нужно создавать конструктором без параметров. **То есть, инициализировать все данные базового класса, а потом уже с помощью конструктора производного класса мы до инициализируем данные производного класса.** А если у нас в базовом классе конструктора без параметров нету, то это значит, что при создании объекта производного класса компилятор не будет знать какой конструктор базового класса вызывать. В этом случае, выход только один: мы должны явно указать компилятору какой конструктор базового класса мы хотим вызывать, и какие параметры передать. Это делается с помощью ключевого слово **base**. Синтаксис:

```
модиф_доступа имя_констр_произв_класса(парам): base(парам)
{…}
```

“Что здесь происходит?” – Конструктор базового класса с параметром инициализирует все, что должно было инициализирована теоретически при создании объекта типа базового класса. И тело внутри конструктора производного класса до инициализирует все остальное. В тот момент, когда мы указываем **base**, мы можем вызывать любой конструктор базового различая их с количеством и типами параметров.

То есть, если в нашем базовом классе есть конструктор без параметров, то компилятор может его вызвать и подставить автоматически. Если у нас нет конструктора без параметров, или если мы не хотим, чтобы компилятор автоматически что-то подставлял, мы должны явно вызывать конструктор базового класса.

Для того, чтобы мы могли использовать конструктор базового класса в производном классе, мы должны объявлять его как **public** или **protected**.

Когда мы объявляем конструктор с модификатором доступа **protected**, мы запрещаем создание объектов этого класса во внешнем коде, но при этом у нас есть возможность создавать производные классы этого класса:

Вызов конструктора базового класса

```
class Token
{
    protected Token(string name) { ... }
    ...
}
class CommentToken: Token
{
    public CommentToken(string name) : base(name) { }
    ...
}
```

ПОДДЕРЖКА ПОЛИМОРФИЗМА(ПЕРЕГРУЗКА МЕТОДОВ)

Если мы хотим перегрузить метод, хотим предоставить механизм поддержки полиморфизма в производных классах, то мы должны объявить в базовом классе все методы, которые мы хотим переопределять в будущем со специальным словом **virtual**. Слово **virtual** нас ни к чему не обязывает. Ключевое слово **virtual** означает, что метод виртуальный, другими словами, этот метод может быть переопределен в дочерних классах, но слово **virtual** не обязывает его переопределять.

ГЛАВА 7. ОПЕРАТОРЫ. ПЕРЕГРУЗКА ОПЕРАТОРОВ

В C# у нас есть возможность переопределять операторы(+, -, /, *, и т.д.) таким образом, чтобы они позволили выполнять соответствующий им операции. Мы можем переопределять операторы в C#. Пример переопределения оператора “+” для типа Time:

```
public static Time operator + (Time t1, Time t2)
{
   int newHours = t1.hours + t2.hours;
   int newMinutes = t1.minutes + t2.minutes;
   return new Time(newHours, newMinutes);
}
```

Если мы хотим переопределить какой-то оператор, то первое, что мы должны сделать это – определить метод класса на статический. Этот метод класса имеет также возвращаемое значение(в нашем примере тип Time), но нам не важно что оно возвращает, потому что мы можем неожиданно прибавить к SuperInt-у int и получить int. После возвращаемого значение идет ключевое слово operator, после этого ключевого слово идет тот оператор, который мы собираемся перегружать(изменять). В нашем примере это “+”. Затем идет параметры. В зависимости от типа оператора параметры разные, параметров может быть несколько. А в нашем примере оператор “+”, и логично, что оператор “+” должен принимать 2 параметра в качестве аргументов (2 слагаемых). Но здесь есть ограничения по отношению к аргументам метода оператора – типы, то есть один из аргументов метода оператора должен иметь тип нашего класса:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace Operators
{
    class Program
    {
        static void Main(string[] args)
        {
            SuperInt s = new SuperInt(5);
            SuperInt s1 = new SuperInt(8);
            SuperInt res = s + s1;
            res.Print();

            Console.ReadLine();
        }
    }

        class SuperInt
        {
            int n;

            public SuperInt(int n)
            {
                this.n = n;
            }

            public static SuperInt operator +(SuperInt s1, SuperInt s2)
```

```
            {
                return new SuperInt(s1.n + s2.n + 5);
            }

            public void Print()
            {
                Console.WriteLine(n);
            }
    }
}
```

Результат: 18

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace Operators
{
    class Program
    {
        static void Main(string[] args)
        {
            SuperInt s = new SuperInt(5);
            SuperInt s1 = new SuperInt(8);
            SuperInt res = s + s1 + 2;
            SuperInt res1 = 2 + s + s1;
            res.Print();
            res1.Print();

            Console.ReadLine();
        }
    }

        class SuperInt
        {
            int n;

            public SuperInt(int n)
            {
                this.n = n;
            }

            public static SuperInt operator +(SuperInt s1, SuperInt s2)
            {
                return new SuperInt(s1.n + s2.n);
            }

            public static SuperInt operator +(SuperInt s1, int s2)
            {
                /* Возвращает сложение значение n, который находится внутри типа
SuperInt и значение int-a, который мы передадим */
                return new SuperInt(s1.n + s2);
            }

            public static SuperInt operator +(int s1, SuperInt s2)
            {
                /* Возвращает сложение значениезначение int-a и n, который находится
внутри типа SuperInt */
                return new SuperInt(s1 + s2.n);
```

```
            }

            public void Print()
            {
                Console.WriteLine(n);
            }
    }
}
```

Результат: 15 15

В методах перегрузки операторов порядок параметров важен. Поэтому если наши операции(операторы) обладают транзитивностью, то необходимо это реализовать явно путем определения соответствующего набора методов для реализации транзитивности (транзитивность, когда важен порядок параметров). “Зачем это нужно?” – Например, для операции “-” нам важен порядок параметров, то есть “s1-s2” это не то же самое, что “s2-s1”.

Если мы хотим написать полноценный тип, который может участвовать в самых сложных выражениях, мы должны все операторы (-, +, /, и т.д.) переопределить. Переопределение возможно для всех операторов, которые выполняют простейшие арифметические операции. Ну есть и некоторые **ограничения**:

1) **Перегрузка операторов сравнения:**

- должны перегружаться попарно:

- **“<” и “>”**
- **“<=” и “>=”**
- **“==” и “!=”.**

Если в сложении этого требование не было, то есть мы могли и не реализовывать попарно методы, а для операторов сравнения нужно выполнять перегрузку методов попарно. Все типы(классы) в C# неявно наследуются от **object**.

2) **С операторами “==” и “!=” следует перегружать метод Equals и GetHashCode**. Методы Equals и GetHashCode в классе object объявлены как virtual, поэтому их стоит перегружать с помощью ключевого слово override.

ПЕРЕГРУЗКА ЛОГИЧЕСКИХ ОПЕРАТОРОВ

Операторы && и || не могут быть перегружены напрямую, потому что && и || разворачиваются в более сложные выражения, которые используют битовые операции, и используют операторы true и false, бинарные true и false. Если мы хотим реализовать какой-то тип, связанный с && или ||, то нам нужно перегружать логические && и ||, то нам придется для их перегрузки перегрузить вот эти 4 оператора: &, |, true, и false.

ПЕРЕГРУЗКА ОПЕРАТОРОВ, СВЯЗАННЫЙ С ПРИВЕДЕНИЕМ ТИПОВ

При приведения типов неявно или явно должно быть где-то перегружено операция приведения. То есть, должно быть где-то прописано, что int можно приводить к double, причем неявно.

А с double к int можно приводить явно. И здесь тоже где-то должно быть прописано, что double можно приводить явно к int. Вот это и делается с помощью перегрузки операторов. Синтаксис:

```
public static explicit operator Time(float hours){}

public static explicit operator float(Time t1){}
```

```
public static implicit operator string(Time t1){}
```

`explicit` – означает, что мы перегружаем явное приведение.
`implicit` – означает, что мы перегружаем неявное приведение типа.

После слов `explicit`, `implicit` идет ключевое слово operator. После слово operator идет тип, к которому мы выполняем приведение нашего типа. А в качестве параметра мы указываем переменную начального типа. Пример:

```
public static implicit operator SuperInt(int n)
{
   return new SuperInt(n);
}
```

В этом примере мы перегружаем метод для того, чтобы неявно привести int к SuperInt-у. А если мы хотим SuperInt привести к int-у(явное приведение):

```
public static explicit operator int(SuperInt s)
{
   return s.n;
}
```

Пример:

```
using System;
namespace Operators
{
    class Program
    {
        static void Main(string[] args)
        {
            SuperInt s = 6; /* Теперь мы можем присваивать к переменным типа SuperInt
переменные типа int, благодаря перегрузке метода. То есть здесь будет неявное
приведение логику, которого мы реализовали */

            int n = (int)s; /* Здесь мы используем реализацию перегрузки метода - явное
приведение */
            Console.WriteLine(n);
            Console.ReadLine();
        }
    }
        class SuperInt
        {
            int n;

            public SuperInt(int n)
            {
                this.n = n;
            }

            public static SuperInt operator +(SuperInt s1, SuperInt s2)
            {
                return new SuperInt(s1.n + s2.n);
            }
```

```
            public static SuperInt operator +(SuperInt s1, int s2)
            {
                /* Возвращает сложение значение n, который находится внутри типа
SuperInt и значение int-a, который мы передадим */
                return new SuperInt(s1.n + s2);
            }

            public static SuperInt operator +(int s1, SuperInt s2)
            {
                /* Возвращает сложение значениезначение int-a и n, который находится
внутри типа SuperInt */
                return new SuperInt(s1 + s2.n);
            }

            // Перегрузка метода для приведение типа int к SuperInt
            public static implicit operator SuperInt(int n)
            {
                return new SuperInt(n); /* Внутри этого мы можем сами определять логику
метода */
            }

            // Перегрузка метода для приведение типа SuperInt к int
            public static explicit operator int(SuperInt s)
            {
                return s.n; // Внутри этого мы можем сами определять логику метода
            }

            public void Print()
            {
                Console.WriteLine(n);
            }
    }
}
```

Результат: 6

Если мы выполняем преобразование к строке (string), то желательно вместе с преобразованием к типу string выполнять перегрузку метода ToString(). Потому что, логично преобразование к типу string и метод ToString() должны возвращать одно и тоже.

Механизм перегрузки операторов мощный механизм, который позволяет сделать свой собственный тип, расширив структуру стандартных типов, таких как: int, double и т.д., при этом наш собственный тип будет полностью подчиняться всем критериям, которые применимы к стандартным типам.

Чтобы создать альтернативное имя для нашего типа, мы используем такую конструкцию:

using альтернативное_имя = имя_простр_имен.имя_нашего_типа;

ГЛАВА 8. СВОЙСТВА И ИНДЕКСАТОРЫ

В классах данные желательно объявлять с модификатором доступа private.

Когда у нас в классе есть данные с модификатором доступа private, и мы хотим предоставить возможность внутри класса делать так, чтобы я мог используя объект класса достаточно просто менять нашу приватную переменную просто установив какое-то значение, с другой стороны, мы должны иметь возможность получить данные(приватные переменные) с помощью какого-то метода. Для того, чтобы устанавливать данные и для того, чтобы получать данные мне нужно объявить 2 метода. И эти методы должны иметь модификатор public. **Первый метод возвращает значение переменной**:

```
public тип_перем. имя_метода()
{
   ...
   return имя_перемен;
}
```

Причем, имя метода желательно писать понятным. "…" – здесь может идти куча каких-то проверок, предварительной проверки, обработки данных, приготовление их к возврату. На самом деле здесь может быть написано много всякого кода. К примеру, если пользователь не имеет права на получение переменной здесь может быть выброшено исключение.

Второй метод позволяет нам установить значение нашей переменной. Он, естественно, принимает какой-то параметр, потому что нам необходимо где-то взять значение, которую мы присвоим переменной:

```
public void имя_метода(тип_параметра имя_параметра)
{
   ...
   имя_перем = имя_параметра;
}
```

Здесь тоже может идти предварительная обработка, проверки и т.д.("…"), и после этого мы используем код по установке нашего приватного поля.

На самом деле, в этих двух методах может быть любой код, но в нашем примере класс простой, поэтому никакого дополнительного кода нет. Пример:

```
namespace Modul13
{
    class Program
    {
        static void Main(string[] args)
        {}
    }

    class MyClass
    {
        private string firstName;

        // Возвращает нашу переменную
```

```
        public string GetFirstName()
        {
            /* Здесь может быть написано много кода, который проверяет, обрабатывает, и
подготавливает их к возвращению */
            return firstName;
        }

        // Позволяет нам установить значение нашей переменной
        public void SetFirstName(string name)
        {
            /* Здесь может быть написано много кода, который проверяет, обрабатывает
переменную */

            // Код по установке нашего приватного поля
            firstName = name;
        }
}
}
```

Имена методов непринципиально указывать по каким-то правилам, но желательно придумывать их, чтобы они раскрывали суть метода. Но каждый разработчик вправе сам придумывать имена этим методам, и сам реализовывать метод, и это приводит к путанице когда другой разработчик разбирается в его коде. Поэтому в C# ввели понятие **свойство**. Фактически свойства позволяют объединить внутри себя возможности, которые предоставляют наши выше рассмотренные 2 метода по доступу к какой-то закрытой переменной класса. С одной стороны, свойство – набор двух методов(выше рассмотренных), с другой стороны, свойство – механизм, который позволяет общаться с закрытой переменной. Синтаксис определения свойства:

```
модификатор_доступа тип_свойство имя_свойство
{
   get{ return имя_перем; }
   set{ имя_перем = value; }
}
```

Обычно модификатор доступа у свойства, это – public. Но допустимы и другие модификаторы доступа.

Если мы объявляем свойства, то уже понятно, что внутри свойства должна быть некая конструкция, которая с одной стороны возвращает какие-то данные указанного типа, и позволяет установить какие-то данные в переменную. Внутри свойства, когда мы описываем методы доступа (наши 2 метода, выше определенных) нам не нужно полностью писать сигнатуру метода, мы можем полностью сигнатуру метода заменить на ключевое слово **get**(в нашем примере вместо GetFirstName) и **set**(в нашем примере вместо SetFirstName). “А откуда теперь брать значение параметра?” – Этот вопрос решается с помощью ключевого слово **value**. То есть, value – параметр, который мы передаем при установке каких-то данных в нашем классе. value будет иметь тип нашего свойства. Пример:

```
namespace Modul13
{
    class Program
    {
        static void Main(string[] args)
```

```
        {}
    }

    class MyClass
    {
        private string firstName;

        // Свойство
        public string FirstName {
        //Возвращает нашу переменную
        get
        {
            /* Здесь может быть написано много кода, который проверяет, обрабатывает, и
подготавливает их к возвращению */
            return firstName;
        }

        // Позволяет нам установить значение нашей переменной
        set
        {
            /* Здесь может быть написано много кода, который проверяет, обрабатывает
переменную. Код по установке нашего приватного поля */
            firstName = value;
        }
    }
    }
}
```

Работа со свойством выглядит снаружи, как и работа с обычной переменной. Пример:

```
using System;

namespace Modul13
{
    class Program
    {
        static void Main(string[] args)
        {
            MyClass c = new MyClass();
            c.FirstName = "Sergey";
            Console.Write(c.FirstName);

            Console.ReadLine();
        }
    }

    class MyClass
    {
        private string firstName;

        // Свойство
        public string FirstName {
        // Возвращает нашу переменную
        get
        {
            /* Здесь может быть написано много кода, который проверяет, обрабатывает, и
подготавливает их к возвращению */
            return firstName;
        }
```

```
        // Позволяет нам установить значение нашей переменной
        set
        {
            /* Здесь может быть написано много кода, который проверяет, обрабатывает
переменную. Код по установке нашего приватного поля */
            firstName = value;
        }
    }
    }
}
```

Результат: Sergey

Используя свойство мы имеем возможность контролировать те данные, которые поступают на вход, и прежде чем их куда-то присваивать мы имеем возможность проверять. И в случае ошибочных данных можно выбрасывать исключение, можно предварительно готовить (изменять) данные перед тем, как их возвращать и т.д.

ПОНЯТИЕ СВОЙСТВ

- работа как с полями, но позволяют выполнять код
- методы доступа
    - использование get метода доступа для получения данных
    - использование set метода доступа для присваивания значение.

```
class Button
{
    public string Caption // Property
    {
        get { return caption; }
        set { caption = value; }
    }
    private string caption; // Field
}
```

ТИПЫ СВОЙСТВ

У нас не всегда есть необходимость выполнять оба действия (get, set): 1) Получать какие-то данные; 2) Устанавливать какие-то данные. То есть, иногда у нас необходимость есть только установить(set) данные. Чаще у нас наоборот, свойство служит механизмом получения данных, то тогда они не позволят установить данные. К примеру, свойство length класса string, оно позволяет получить количество символов в строке, но оно не позволяет их установить, потому что строка – такой объект, который не мутирует(является фиксированной). Поэтому, у нас должен быть механизм, который записывает свойства, таким образом, что они позволяли делать только одно из действии(get или set).

Если мы не хотим, чтобы свойства давала доступ к данным, мы можем удалить полностью блок **get**. И теперь это свойство позволяет только устанавливать данные. Пример:

```
using System;
```

```
namespace Modul13
{
    class Program
    {
        static void Main(string[] args)
        {
            MyClass c = new MyClass();
            c.FirstName = "Sergey";
            Console.Write(c.FirstName); /* Теперь эта строка ошибочная, так как
FirstName не позволяет вернуть данные, позволяет только устанавливать */

            Console.ReadLine();
        }
    }

    class MyClass
    {
        private string firstName;

        // Свойство
        public string FirstName {

        // Позволяет нам установить значение нашей переменной
        set
        {
            /* Здесь может быть написано много кода, который проверяет, обрабатывает
переменную. Код по установке нашего приватного поля */
            firstName = value;
        }
    }}}
```

А если мы не хотим устанавливать, а только получать доступ к данным(переменной), то мы удаляем блок **set**. Пример:

```
using System;

namespace Modul13
{
    class Program
    {
        static void Main(string[] args)
        {
            MyClass c = new MyClass();
            c.FirstName = "Sergey"; /* Теперь эта строка ошибочная, так как FirstName
не позволяет устанавливать данные, а позволяет только получать */
            Console.WriteLine(c.FirstName);

            Console.ReadLine();
        }
    }

    class MyClass
    {
        private string firstName;

        //Свойство
        public string FirstName {

            get
            {
                return firstName;
```

```
            }
    }
}
}
```

Все методы доступа – get и set имеют модификатор доступа свойства, то есть public. Но если мы хотим для некого метода доступа понизить модификатор доступа, то это вполне возможно. К примеру, если мы хотим получать данные, но устанавливать данные мы хотим только внутри класса, тогда мы явно прописываем модификатор доступа у метода доступа set как private. Здесь может быть любой модификатор для разных случаи. Мы также можем явно прописывать модификатор доступа public.

Модификатор доступа у свойств может быть разный: если есть необходимость сделать свойства только для чтения для внешнего мира, это не значит, что надо убирать полностью метод set. Это значит, что его надо сделать protected или private.

Когда нам не нужно дописывать дополнительный код в метод get и в метод set, то тогда мы можем использовать облегченный синтаксис для свойств:

```
модификатор_доступа тип_свойства имя_свойства
{
  get;
  set;
}
```

Естественно, здесь для каждого метода можем писать разные модификаторы доступа. Пример:

```
public string FirstName{private get; set;}
```

Поэтому, если мы не собираемся писать дополнительный код для get и set, то подобный способ описаний свойств достаточно удобный. Ну а если мы уже решились написать какой-то код в get и set, и поскольку в коде мы работаем через свойства, мы раскрываем метод get и set, прописываем какую-то приватную переменную и начинаем работать. То есть, фактически мы просто реализуем это свойство более детально.

ТИПЫ СВОЙСТВ

- Read/Write свойства
  - присутствуют get и set методы доступа
- Read-only свойства
  - имеют только get метод доступа
- Write-only свойства
  - имеют только set метод доступа
- Static свойства
  - могут получать доступ только к статическим данным

- возможны различные модификаторы доступа для get и set

Свойства могут быть static, тогда эти свойства будут доступны на уровне класса. К примеру, **Console.Title** и т.д. Статические свойства позволяют получить доступ только к статическим данным. И доступ к статическому свойству осуществляется через имя класса, то есть все то же самое, что и для статических данных.

Во всех вышесказанных нами свойствах мы перед свойством объявляли поля и использовали его в свойстве. Тут ошибочно может сложиться ощущение, что свойство всегда обязана быть связана с какими-то данными. На самом деле, это не так. Могут быть свойства и эти свойства могут быть абсолютно не связаны с данными. К примеру:

```
public string FirstName
{
  get
  {
    return “Sergey”;
  }
}
```

Другими словами, свойства и данные не обязаны быть между собой связаны. То есть, свойство – механизм, который вроде бы позволяет куда-то отправить данные и позволяет вернуть какие-то данные.

ПОНЯТИЕ ИНДЕКСАТОРА

Как и свойство, индексатор предоставляет доступ к каким-то данным внутри объекта класса. Но при этом, он это делает ни через имя, а он это делает используя синтаксис массива, то есть используя стандартные квадратные скобочки для определения индекса. Иными словами, индексатор – механизм, который позволяет для объекта нашего класса писать вот что-то такое:

имя_объекта_класса[индекс] = значение;

Индексаторы позволяют индексировать экземпляры класса или структуры так же, как и массивы.

Синтаксис индексатора:

```
public тип_возвращаемого_значения this[параметры]
{
  get{ return значение; }
  set{ … }
}
```

Явно имени у индексатора нету, поэтому используем ключевое слово `this`. **тип_возвращаемого_значения** – тип, который должен быть у значении членов

индексатора. В синтаксисе определения индексатора ключевое слово this говорит, что мы перегружаем оператор квадратные скобочки. А после ключевого слово `this` указываются параметры. Поскольку мы перегружаем оператор квадратные скобки, у нас должно быть 2 метода. Один метод возвращает какие-то данные используя индексатор, второй метод позволяет присвоить какие-то данные. Очевидно, здесь у нас тоже должно быть 2 метода доступа: `get` и `set`. Пример:

```
public int this[int index]
{
  get { return 0; }
  set {  }
}
```

В индексаторе в квадратных скобках мы указываем индекс(параметр). И когда мы будем возвращать или устанавливать значение мы будем опираться на этот параметр. То есть, с помощью этого индекса мы находим в классе нужное место, и устанавливаем переменную, и эта переменная храниться в `value`.

В методе get мы можем использовать параметр index, а в методе set мы можем использовать и параметр index, и параметр value(index определяет куда нам надо вставить, а value определяет значение, которая передается при присваивание):

```
public int this[int i]
{
  get
  {
    return arr[i];
  }
  set
  {
    arr[i] = value;
  }
}
```

“Какого типа должен быть индекс?” – **Любого**. То есть, индекс может быть любого типа, даже типа String, Char, double. В обычных массивах, списках индекс обычно типа int. ”К примеру, как создать многомерные(двумерные, трехмерные, и т.д.)?” – Мы просто будем увеличивать количество параметров(сколько нужно):

```
public int this[int index, int index2 …]
{
  get { return 0; }
  set {  }
}
```

Пример:

```
using System;

namespace Modul13
{
    class Program
    {
        static void Main(string[] args)
        {
            MyList list = new MyList();

            Console.WriteLine(list[3]);
        }
    }

    class MyList
    {

        Node root;  // Первая Node-а в нашем списке

        public class Node
        {
            public int Data { get; set; } // Свойство тип int

            public Node Next { get; set; } // Свойство, которая ведет на следующию
Node-у
        }

        public int this[int index]
        {
            get
            {
                return 0;
            }
            set { }
        }
    }
}
```

Причем, индексы могут иметь разные типы. При вызове нашего множества, мы будем указывать индексы через запятую.

То есть по большему счету, индексатор – перегрузка оператора квадратные скобки. Определение индексатора похоже на определение свойства, единственное, что вместо имени используются ключевое слово this и квадратные скобки, которые принимают любое количество параметров и любых типов. А доступ осуществляется стандартным способом:

имя_объекта_класса[парам.];

Индексатор подчиняется правилам перегрузки, то есть меняя количество, тип, и порядок параметров мы получаем разные индексаторы. Пример:

```
using System;

namespace Modul13
{
    class Program
```

```
    {
        static void Main(string[] args)
        {
            MyList list = new MyList();

            Console.WriteLine(list[3]);
        }
    }

    class MyList
    {

        Node root;  // Первая Node-а в нашем списке

        public class Node
        {
            public int Data { get; set; } // Свойство тип int

            public Node Next { get; set; } // Свойство, которая ведет на следующию
Node-у
        }

        public int this[int index, char index2]
        {
            get
            {
                return 0;
            }
            set { }
        }

        public int this[char index2, int index]
        {
            get
            {
                return 0;
            }
            set { }
        }

        public int this[int index]
        {
            get
            {
                return 0;
            }
            set { }
        }
    }
}
```

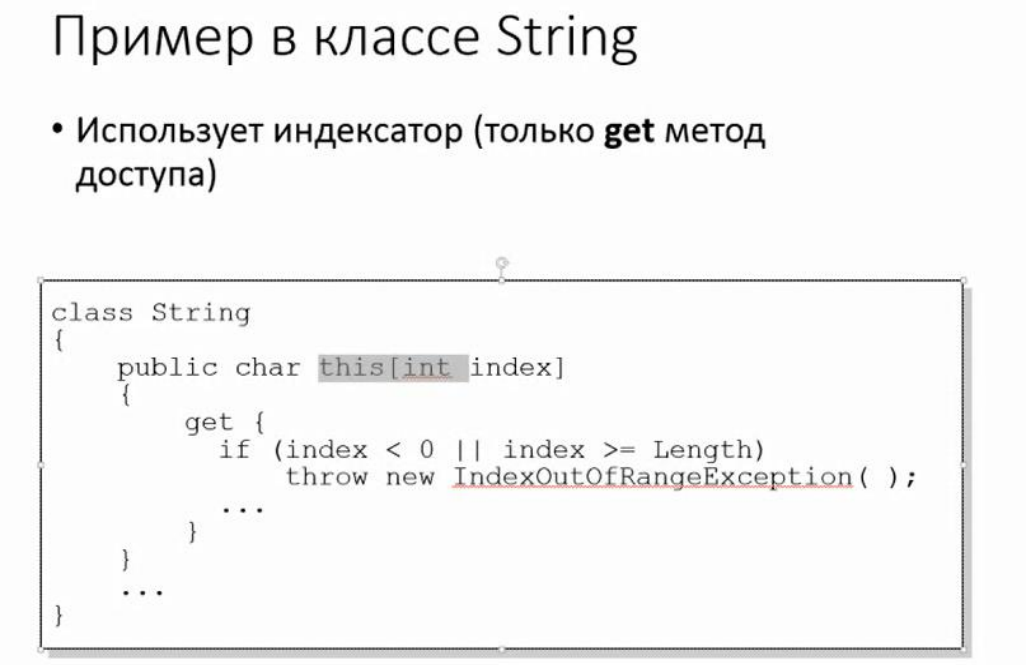
Индексаторы используются, если мы хотим делать списки, графы, и т.д.

ПОНЯТИЕ ИНДЕКСАТОРА

- доступ к данным в формате массивов
- возможность использовать индексы любого типа и различное количество параметров
- для создания индексатора

- создание свойства с именем this
- определение типов индексов
- использование индексаторов
  - формат массива
- поддерживает перегрузку

Пример индексатора в классе String: Если мы определяем переменную типа String мы можем получить доступ к любому символу в этом типе. Пример:



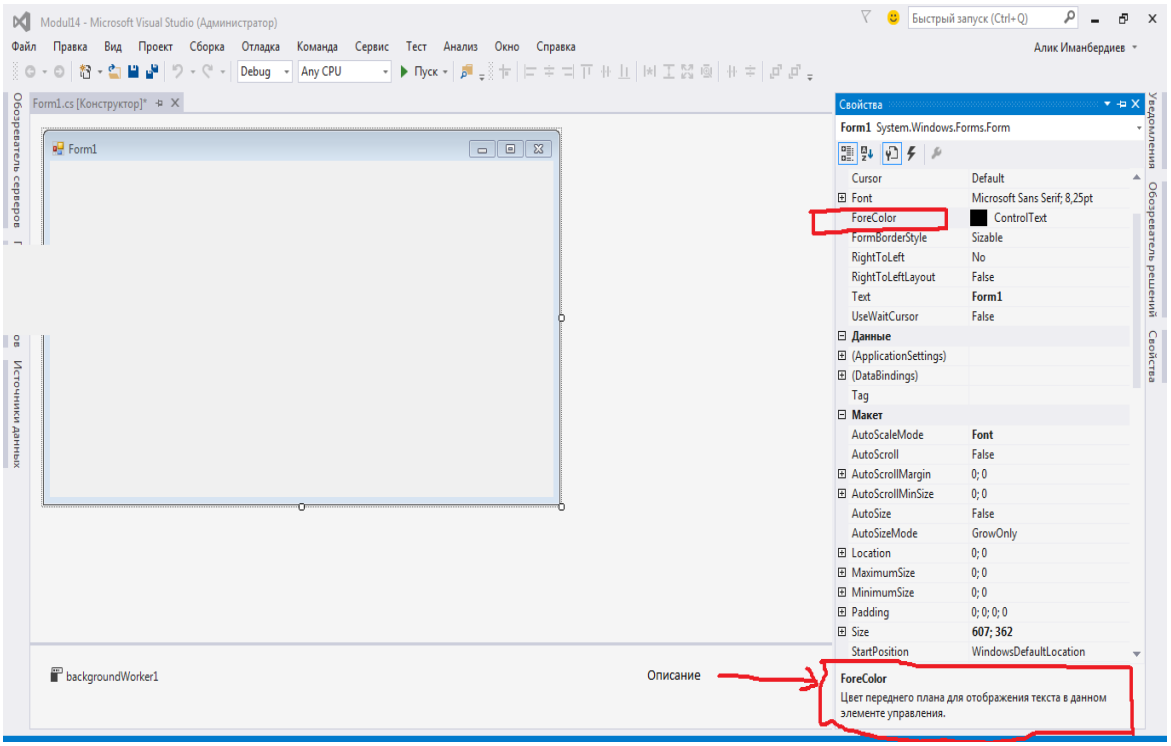
ГЛАВА 9. АТРИБУТЫ

“Можно ли внутри сборки поместить свою метаинформацию?” – Можно. И эта как раз делается с помощью атрибутов. Атрибуты позволяют сохранять информацию внутри приложения, которая может быть доступна во время выполнения. Причем, в общем-то эта метаинформация может никоим образом не влиять на работу самого приложения. То есть, если у нас есть классы, которые описывают наши формы, описывают нашу бизнес логику, то есть классы нашего приложения, а когда мы захотим внедрить внутрь нашего приложения дополнительную метаинформацию, которая может использоваться, может не использоваться. Причем, если мы хотим внедрить эту метаинформацию, мы можем внедрить ее не только на уровне сборки, не только привязать ее к самому приложению, но мы также можем ассоциировать ее с каким-то классом, с какими-то методами, с какими-то полями данных. Все это можно сделать с помощью атрибута. То есть, фактически атрибут – некоторый объект, который встраивается внутрь нашего работающего приложения. Поскольку, это – объект, то он создается на основе какого-то класса, где есть какой-то конструктор, то есть он как-то сконструирован будет во время запуска нашего приложения. И он будет находится внутри нашего приложения, при этом не будет иметь особого влияния на работу приложения. Но при этом к этому объекту может всегда обратится и само приложение, и возможно какие-то сторонние приложение(если, к примеру, речь идет не об .exe файле, а о библиотеке).

ПОНЯТИЕ АТРИБУТОВ

- возможность сохранять информацию в приложении, которая может быть доступна во время выполнения
- данные могут быть привязаны к большинству элементов в коде
    - приложению
    - классу
    - методам и полям данных
- атрибуты могут быть:
    - определены разработчиком
    - использоваться из существующего набора в .Net Framework.

Пример атрибута:



В Windows Forms Application свойства элементов имеют описание(мы выделили в вышеуказанном примере). Вот эти описание Visual Studio и берет из сборки в которой лежит элемент, в которой лежит описание класса элемента. И эти описания(выделенный в примере) как раз и сохраняются внутри элемента с помощью атрибутов. В примере, в WFA в элементе Form атрибут – Description. Visual Studio когда мы добавляем очередной элемент на форму, Visual Studio обращается к сборке, где хранится этот элемент, и пытается получить информацию о всех атрибутах, которые ассоциированы со свойствами, отображаемых в окне Properties. То есть, с одной стороны здесь(в окне Properties) выводится все свойства, с другой стороны здесь с помощью атрибутов выводится дополнительная информация. К примеру, описание, также есть масса другой дополнительной информации, которая задается с атрибутами(какой редактор использовать для редактирования того или другого атрибута, как преобразовывать данные). В принципе такой атрибут как Description нигде в приложение не используется(он просто выводит информацию).

Атрибут – это класс. То есть, атрибут – это класс, который построен по определенному образу, и объекты которого можно внедрять в работающие приложения, причем, связывать их с какими-то компонентами нашего приложения(с классами, с методами, со свойствами, и с полями данных).

“Как используются атрибуты?” – Во многих кодах внутри кода используется какая-та запись – квадратные скобки, и внутри квадратных скобок пишется имя класса, параметры передаются. Это и есть описание атрибута:

[attribute(positiona;_parameters, named_parameter = value, …)] element
• к одному элементу может быть привязано несколько атрибутов

В Visual Studio(в консольном приложении) атрибуты хранятся в Properties -> AssemblyInfo.cs. Эти атрибуты позволяют нам назначить метаданные для сборки. Эти атрибуты позволяют нам задать версию сборки, эти атрибуты, которые уникальный идентификатор, культуру, описание, заголовок, имя компании и т.д. Эти атрибуты предопределены в .Net Framework и мы их можем использовать при описании нашей сборки. Формат сборки: квадратные скобки, внутри квадратных скобок пишется имя класса, описывающий тот или другой атрибут, далее в круглых скобках передаются параметры(то есть, это – стандартный вызов конструктора). Атрибуты связывают с какими-то элементами нашего приложения. Если хотим, атрибут связать с каким-то классом, то вам необходимо его писать непосредственно перед самим классом, а если бы атрибут был привязан с методом, то его надо было бы писать непосредственно перед методом в квадратных скобках указав имя атрибута, и передав параметры(в случаях полей тоже так). Ну а если мы привязываем к сборке, то мы должны использовать приставку “**assembly:**”. “**assembly:**” означает, что вы далее собираетесь написать(внедрить) атрибут в приложения, который будет ассоциироваться со сборкой в целом, то есть не будет привязан к какому-то классу, свойству, а он будет ассоциироваться именно с целой сборкой.

ГЛАВА 10. ДЕЛЕГАТЫ И СОБЫТИЯ

В C# делегат помимо того, что позволяет задать тип, создающий динамический массив, хранящий ссылки на методы внешних объектов, он позволяет еще задать тип этих методов. Каждый делегат задает свой конкретный тип метода. К примеру:

```
delegate string TableNotificationDelegate();
```

В этом примере создается тип с именем **TableNotificationDelegate**, и этот тип способен хранить целый набор ссылок на методы типа возвращающих string и не принимающих параметры. То есть, делегат **TableNotificationDelegate** может указывать на методы, которые не принимают параметры и возвращают string. Делегат – это класс. Причем класс, который наследуются автоматически от класса MulticastDelegate. Синтаксис делегата:

```
модификатор_доступа delegate возвращаемый_тип имя(список_параметров);
```

где **возвращаемый_тип** обозначает тип значения, возвращаемого методами, которые будут вызываться делегатом; **имя –** конкретное имя делегата; **список_параметров** – параметры, необходимые для методов, вызываемых делегатом. То есть, **синтаксис делегата должен соответствовать синтаксису метода, на которую он указывает**.

Делегат – класс, который может содержать ссылки на методы определенной сигнатуры. И далее мы можем использовать этот класс(делегат), чтобы внутри других классов объявить переменную типа делегата, которая будет содержать ссылки на наши методы. Синтаксис:

```
модификатор_доступа имя_делегата имя_переменной;
```

С помощью **имени_переменной типа_делегата** ????, мы можем вызывать все методы, которые хранятся внутри **имени_переменной((объекта)**. То есть, мы просто можем написать **имя_переменной**, передать нужные параметры, и она автоматически дергает всю цепочку методов, которые хранятся внутри этой **имени_переменной(объекта)**.

Ссылка(имя_переменная) на объект позволяет хранить методы со сигнатурой, указанном нами в делегате формате.

Если у нашего метода не соответствует сигнатура с сигнатурой делегата, а мы хотим вызывать(привязать) этот метод с помощью нашего делегата, то мы можем создать еще один метод с сигнатурой как у делегата, внутри этого метода вызвать наш метод(то есть, у которого не совпадает сигнатура). И привязать этот метод(у которого сигнатура совпадает с делегатом) к делегату.

Для того, чтобы привязывать методы к делегату, то есть чтобы делегат указывал на эти методы нам нужно в момент создания объекта делегата в скобках указать метод, на который будет указывать делегат:

имя_делегата имя_объекта = new имя_делегата(имя_метода);

Когда мы используем операцию “=”, то мы перезаписываем значение объекта(**имя_объекта**) делегата.

Для того, чтобы делегат указывал на несколько методов сразу, мы должны добавлять методы к объекту делегата с помощью оператора “+=”:

имя_делегата имя_объекта += new имя_делегата(имя_метода);

А если мы хотим наоборот убирать методы, то мы используем оператор “-=”:

имя_делегата имя_объекта -= new имя_делегата(имя_метода);

Делегат предоставляет мощный способ хранение ссылок на внешние методы. И с помощью этого делегата мы можем вызывать эти методы.

В терминах C# механизм событии позволяет реализовать именно делегаты. Ключевое слово `event` позволяет применить хорошую практику для написания объектов типа делегата. Синтаксис события:

модификатор_доступа `event` **делегат_события имя_события;**

Пример:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace LABKA
{
   /* Обьявление делегата, указывающий на методы не принимающих параметры и
возвращающих void */
    delegate void dDelegate();

    class Program
    {
        static void Main(string[] args)
        {
            A a = new A();
            B b = new B();

            // Привязываем к событию Per метода Show и ShowP
```

```
            a.Per += new dDelegate(a.Show);
            a.Per += new dDelegate(b.ShowP);


            // Привязываем к обьекту делегата h метод Show
            dDelegate h = new dDelegate(a.Show);
            h += b.ShowP;
            h();
        }
    }

    class A
    {
        // Событие
        public event dDelegate Per;


        public void Show()
        {
            Console.WriteLine("Method Show");
        }
    }

    class B
    {
        public void ShowP()
        {
            Console.WriteLine("Method ShowP");
        }
    }
}
```

Делегат – динамическая структура, которая хранит в себя ссылки на методы. И с помощью этого делегата мы можем вызывать эти методы.

ГЛАВА 11. ОБОБЩЕННЫЕ ТИПЫ (GENERICS)

Обобщенный тип позволяет при создании объекта класса ввести дополнительные параметры. Дополнительные параметры - это не параметры конструктора, а параметры которые будут передавать объекту класса некоторый тип, который класс может внутри себя использовать в качестве шаблона. **Обобщенные типы** являются частью .Net Framework, и поэтому классы, которые построены на основе обобщенных типов(Generics), их можно размещать внутри библиотеки.

Обобщенные типы позволяют нам при конструирование объектов класса явно передавать обобщенный тип.

При описание класса, интерфейса обобщенный тип надо указывать после имени класса, интерфейса:

```
class имя_класса<T> {…}
```

Когда мы вводим описание типа такого: **имя_класса<T>**, мы подразумеваем, что в последующем все ссылки на класс мы должны описывать в подобном формате **имя_класса<T>**. То есть, мы должны четко указывать какой тип у них будет. То есть, мы должны при создании любой ссылки, объекта этого класса явно указывать обобщенный тип **T** тип. Пример:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace Modul16
{
    class Program
    {
        static void Main(string[] args){}
    }

    interface INode<T>
    {
        T GetData();
    }

    class Tree
    {
        class Node<T> : INode<T>
        {
            T value;

            Node<T> left;
            Node<T> right;

            public T GetData()
            {
                return value;
            }
        }
    }
}
```

Если мы не знаем в классах, интерфейсах, и т.д. каким будет тип, или хотим чтобы тип был любым, мы вводим параметр(**T**), то есть мы тем самым параметризируем тот класс, интерфейс, где будет использоваться наше имя. То есть, используя обобщенный тип мы делаем универсальным наш класс, интерфейс.

ПОНЯТИЕ ОБОБЩЕННЫХ ТИПОВ

- Позволяют создавать интерфейсы и классы с типами в качестве параметров:

```
public class CustomList<T>
    {
        public T this[int index] { get; set; }
        public void Add(T item) { ... }
        public void Remove(T item) { ... }
    }
```

- Указывайте ваш тип при создании объекта класса:

```
CustomList<Coffee> coffees = new CustomList<Coffee>();
```

Естественно, если у нас есть внутренние классы, то мы их можем параметризировать типами, которые приходят на вход внешним классам. Естественно, если мы имеем параметризированный интерфейс, то мы можем его параметризировать тем типом, который приходит на вход классу, который реализует этот интерфейс.

Создание объекта подобного класса(с обобщенным типом):

имя_класса<тип> имя_объекта = new **имя_класса<тип>();**

ПРЕИМУЩЕСТВА ОБОБЩЕННЫХ ТИПОВ

- гарантия наличия данных нужного типа
- нет преобразования
- нет boxing и unboxing механизма.

Когда мы используем обобщенный тип(T), мы ничего о типе T не знаем. То есть, мы не знаем, если у него конструктор; мы не знаем какие операции он поддерживает; какие интерфейсы реализует. Мы вообще об этом типе T ничего не знаем. И когда нам нужно иметь тип, который будет гарантировано поддерживать определенные типы конструкторов, гарантировано поддерживать определенные операции, реализовывать определенные

интерфейсы, и когда нам нужно накладывать ограничение на наш тип. И мы можем реализовать это:

КАК ЗАДАВАТЬ ОГРАНИЧЕНИЯ НА ПАРАМЕТРЫ

Возможно следующие варианты:
where T: <name_of_interface>
where T: <name_of_base_class>
where T: new()
where T: struct
where T: class

После типа **T** прописываем одну или несколько конструкции указываемых в нашей таблице, которые позволяют задать параметры типа **T**. Это делается с помощью ключевого слово **where**, после которого идет правило. К примеру:

1)

where T: <name_of_interface>

И эта запись означает, что все типы, которые будут вместо T реализовывать интерфейс, который мы указываем.

2) А когда мы пишем **where T: <name_of_base_class>**, то мы можем четко гарантировать, что наш тип будет наследником от какого-то базового класса.

3) **where T: new()** позволяет сообщить о том, что тип **T** будет иметь конструктор без параметров. То есть, с помощью оператора **new()**, мы можем задать вариации конструктора, который будет поддерживать тип **T**.

4) С помощью ключевых слов struct и class, мы можем определить чем является тип T. Является ли это переменной значения, то есть позволит ли он создавать переменные значения, или тип описывается классом. То есть, структура или класс:

where T: struct
where T: class

Это может быть важно, потому что в некоторых случаях, мы можем решить, что мы работаем только с переменными значения, и объекты классов принимать не захотим.

То есть, с помощью ключевого слово where, мы можем описывать тип T так, как нам хочется.

Ключевое слов where(ограничение) мы прописываем после типа **T**. Пример:

```
class Bank<T> where T : class
{
   ...
}
```

5) **where T: имя_класса.** Здесь мы говорим, что используемый тип **T** обязательно должен быть классом **имя_класса** или его наследником. Пример:

```
class Bank<T> where T : Account
{
    T[] accounts;

    public Bank(T[] accs)
    {
        this.accounts = accs;
    }
    // Вывод информации обо всех аккаунтах
    public void AccountsInfo()
    {
        foreach(Account acc in accounts)
        {
            Console.WriteLine(acc.Id);
        }
    }
}
```

ГЛАВА 12. РАБОТА С ФАЙЛАМИ И ПОТОКИ

Все классы, которые позволяют нам работать с потоками, и которые позволяют нам работать с файлами находятся в пространство имен **System.IO**.

System.IO – основное пространство имен для работы с файлами, потоками и др.

Первый класс о котором хотелось поговорить, и который находится в System.IO, это класс **File**. Класс **File –** это класс, который объединяет внутри себя множества статических методов, именно этот класс(File) позволяет вам делать простейшие операции с файлами. Первое, естественно, считать какие-то данные, то есть класс File позволяет считывать какие-то данные, причем двумя способами: 1) Вычитать весь текст(здесь речь идет о текстовых файлах). Метод **ReadAllText(…)** позволяет получить фактически готовую строку, которая содержит всю информацию с текстового файла. Этот способ неоптимальный. Поэтому, мы используем метод **ReadAllText(…)** когда требуется считать небольшое количество текста, когда файл не очень большой. А когда речь идет о чтении более сложных данных, то есть когда мы работаем с большими файлами, то мы должны будем работать с потоками.

А второй метод **ReadAllLines(…)** используется когда наш файл имеет строки. Этот метод позволяет нам сразу вычитать массив этих строк. В наших текстовых файлах имеются специальные символы, которые не имеют отображение, но которые означают конец строки, конец файла и т.д. И на эти символы полагаются наши методы когда считывают весь текст, или строки, и т.п.

Также у нас существует методы для записи данных в файл:

1) **WriteAllText(…)** позволяет записать текст в указанный файл, и если файл существует, и если файл уже содержит какой-то текст, то этот метод позволяет просто все перезаписать.

2) **AppendAllText(…)** позволяет добавить какое-то количество текста в конец файла. То есть, если мы знаем, что файл существует, то мы можем расширять его с помощью этого метода.

Когда мы используем методы ReadAllText() и ReadAllLines(…) мы в круглых скобках в качестве параметра указываем путь до файла. А когда мы используем методы **WriteAllText(…)** и **AppendAllText(…)** мы в круглых скобках в качестве параметров указываем путь до файла, а также текст:

File: Чтение данных

- System.IO – основное пространство имен для работы с файлами, потоками и др.
- Чтение данных из файла
  - ReadAllText()
  - ReadAllLines(…)
- Запись данных в файл
  - WriteAllText()
  - AppendAllText()

Кроме методов чтения и записи, класс File содержит множество методов. То есть, класс File позволяет манипулировать файлами, то есть класс File позволяет удалять файл, позволяет проверять существует ли тот или другой файл, получать различного рода информацию о файле. Все это делается с помощью статических методов класса File, и все

эти методы, естественно, принимают в качестве параметров путь к файлу а в более сложных случаях еще и дополнительные параметры.

Когда у нас возникает желание не просто использовать набор статических методов, а работать с файлом как с неким объектом, который можно передавать как параметр каким-то методам, то тогда мы можем использовать альтернативный класс к классу File, это – класс FileInfo. Если класс File обладает статическими методами каждый из которых как минимум требует путь к файлу, то FileInfo позволяет создать объект этого класса(то есть =, класса FileInfo) единожды во время конструирования этого объекта передав полный путь к файлу, а затем с этим объектом мы можем делать дальше дополнительные действия(получить информацию о файле, производить запись, копирование, удаление файла и т.д.).

Класс File и FileInfo – близнецы, только класс File – набор статических методов, а класс FileInfo позволяет делать все то же самое через объект этого класса:

МАНИПУЛЯЦИЯ С ФАЙЛАМИ

- Использование класса File:

```
File.Delete(...);
bool exists = File.Exists(...);
DateTime createdOn = File.GetCreationTime(...);
```

- FileInfo позволяет работать с файлом, как с объектом:

```
FileInfo file = new FileInfo(...);
string name = file.DirectoryName;
bool exists = file.Exists;
file.Delete(...);
```

Аналогично к классу File и к классу FileInfo в .Net Framework существуют классы по работе с каталогами или директориями. Это – класс Directory и класс DirectoryInfo. Здесь все то же самое, но есть возможность создавать каталоги, есть возможность получать список всех файлов с помощью метода **GetFiles()**. В некоторых типах приложения подобные классы(File, FileInfo, Directory, DirectoryInfo) использовать достаточно тяжело, к примеру в Windows Phone или в Windows 8 приложениях, потому что напрямую нельзя работать с дисковой подсистемой, потому что там есть ограничения связанный с доступом к файлам, которые создают пользователи, и там обычно для работы с файлами используются другие наборы классов(storage API). Для десктопных приложении подобные классы(File, FileInfo, Directory, DirectoryInfo) продолжают работать нормально.

Последний класс из этой серии – “**утилитный класс Path**”. Этот класс позволяет получить ряд каких-то вспомогательных информации, то есть он содержит набор утилитных методов, которые не вошли ни в класс File, ни в класс Directory. К примеру, метод **GetExtension()**, который позволяет получить расширение файла. Класс **Path** содержит набор статических методов.

УТИЛИТНЫЕ ФУНКЦИИ КЛАССА Path

```
string settingsPath = "...could be anything here";
//Check to see if path has an extension
bool hasExtension = Path.HasExtension(settingsPath);
//Get the extension from the path
string pathExt = Path.GetExtension(settingsPath);
//Get path to temp file
```

```
string tempPath = Path.GetTempFileName();
```

Все вышесказанные классы(File, FileInfo, Directory, DirectoryInfo, Path) чаще применяются когда мы хотим манипулировать файлами не посредственно. Но когда сы хотим работать с файлами, особенно с большими файлами, мы должны переходить на уровень **потоков**.

Поток – базовое понятие операционной системы, и фактически поток можно охарактеризовать как некий объект, который позволяет нам взаимодействовать с каким-то источником данных. Причем в Windows это понятие ассоциируется не только с файлом. В .Net Framework можно выделить 3 основных типов потоков:

1) **Файловый поток(FileStream)** – объект, который позволяет нам взаимодействовать с файлом. При взаимодействии с файлом мы получаем доступ к некому Handler-у с помощью которого мы начинаем читать данные с файла, причем читать данные с файла байт за байтом, и затем как-то с ними взаимодействовать.

2) **Взаимодействие с памятью – MemoryStream**. MemoryStream используется когда у нас большие объемы памяти и когда мы хотим взаимодействовать с ними на побайтовой основе.

3) И третий тип потока – механизм взаимодействие с данными, которые находятся в сетевом диске, то есть какое-то удаленное хранилище – **NetworkStream**.

Вышеперечисленные 3 типа потока предоставляют механизмы доступа к данным и предоставляют возможность выполнять какие-то действия(читать, записать(байты), и т.д.).

Чтобы облегчить нам работу с потоками в .Net Framework созданы несколько утилитных классов: 1) **StreamReader**, 2) **StreamWriter**, 3) **BinaryReader**, 4) **BinaryWriter**. Эти классы созданы для того, чтобы облегчить работу с потоками, и не важно файловый поток, или поток по работе с памятью(**MemoryStream**). **StreamReader**, **StreamWriter** позволяют работать с текстовыми данными. И фактически файловый поток сами преобразуют набор текстовых строк в байты, из байтов делают текстовые строки, то есть полностью облегчают нам работу. Аналогично существуют 2 класса – **BinaryReader**, **BinaryWriter**. Они служат для работы именно с бинарными данными, но там существуют ряд утилитных методов, которые облегчают работу по работе с бинарными данными. И в первую очередь, это связано с преобразованием различного типа данных в бинарную запись, и наоборот.

ПОНЯТИЕ ПОТОКОВ В .Net Framework

- Доступ к источникам данныз различных типов:

FileStream	Работа с файлом
MemoryStream	Работа с памятью
NetworkStream	Работа с сетью

- Работа с данными определенных типов:

StreamReader	Чтение текстовых данных
StreamWriter	Запись текстовых данных
BinaryReader	Чтение бинарных данных
BinaryWriter	Запись бинарных данных

Пример работы с файловым потокам:

Объект типа **FileStream** создается достаточно просто в качестве параметра мы получаем доступ к файлу, с которым мы хотели бы работать, существует также второй параметр, который определяет что делать, ,если там файла нету и т.д.:

```
string filePath = "C:\\fourth\\applicationdata\\settings.txt";

//Underlying stream to file on the file system
FileStream file = new FileStream(filePath);
```

На основе файлового потока мы можем создать объект `BinaryReader`, который позволит вычитывать данные в бинарном формате. Аналогично мы можем создать объект `BinaryWriter`:

```
// BinaryReader object exposes read operations on the underlying FileStream object
BinaryReader reader = new BinaryReader(file);
```

```
// BinaryReader object exposes write operations on the underlying FileStream object
BinaryWriter writer = new BinaryWriter(file);
```

В принципе конструкторы этих всех классов достаточно простые, но в зависимости от задачи мы можем использовать те или другие методы, чтобы получить доступ к этим данным.

То есть, поток нужен чтобы эффективно работать с файлом:

```
string filePath = "C:\\fourth\\applicationdata\\settings.txt";

//Underlying stream to file on the file system
FileStream file = new FileStream(filePath);

//StreamReader object exposes read opeations on the underlying FileStream object
StreamReader reader = new StreamReader(file);

//StreamWriter object exposes write opeations on the underlying FileStream object
StreamWriter writer = new StreamWriter(file);
```

Потоки используются для работы с файлами. То есть, когда нам нужно провести какие-то операции над файлами, и когда у нас есть много классов, методов, которые позволяют за тебя эти операции сделать. Так вот все эти классы, методы будут принимать в качестве параметра ссылку на поток, обычно на базовый класс **Stream**, потому что этим методам все равно с каким потоком работать. Поэтому вместо класса **File** часто используются потоки.

Потоки – это объекты по доступу к файлу, хранилище в памяти или к удаленному хранилище.

Потоки – механизм получения доступа к данным внутри какого-то хранилище.

ПОНЯТИЕ СЕРИАЛИЗАЦИИ

Генерация данных в определенном формате на основе объектов класса, это и есть **сериализация**. То есть, мы берем объект, и мы это объект преобразуем либо в набор байт(**бинарная сериализация**), либо мы записываем наш объект в **xml** – документ(представляем его структуру, описываем что и где в нашем объекте должно находится), и в случае необходимости мы можем восстановить эти объекты. На самом деле сериализовывать объекты можем абсолютно любым способом, то есть мы можем придумать свой протокол согласно которому мы будем выполнять сериализацию. То есть, берем объект и каким-то образом складываем в файл.

При сериализации мы берем объект и представляем его в неком другом виде, обычно в виде удобным для хранения на диске, для отправки через сеть и т.д. К примеру, у нас есть класс **Person**, и у нас есть объект типа **Person**. И для того, чтобы отправить объект типа Person, мы должны преобразовывать его в какой-то формат (XML, JSON, и т.д.). Наиболее 2 популярных формата, которые используются для передачи данных XML и JSON. То есть объект класса напрямую не передать, ее надо сначала сериализовать. К примеру, преобразовать в XML. А XML по большому счету текст(строка). И мы отправляем эту строчку куда-то на сервер, на сервере берут эту строчку и выполняют обратный процесс. Обратный процесс называется десериализация. На сервере на основе этого XML (JSON, т.д.) формируется уже объект, точно такой же объект, который был у нас на клиенте. Если объект нам передать тяжело, то XML (текст) мы легко можем отправлять. Это – первый случай когда нам нужно использовать сериализацию. Второй случай – это сохранение каких-то временных данных. К примеру, у нас есть какие-то объекты, а когда пользователь выходит из приложения при этом объекты на сервер не ушли (нет соединение), мы выполняем сериализацию этих объектов на диск, то есть мы их преобразовываем в формат XML (JSON и т.д.) и сохраняем на диск. Когда мы получаем XML (JSON и т.д.), и чтобы на основе этого XML создать объекты нам необходимо понимать структуру. Благо XML содержит описание всего этого, поэтому мы можем в принципе сгенерировать автоматически какие-то объекты, они может быть не будут похоже на исходные варианты, но даже если мы не знаем структуру самого XML (точнее природу объектов, которые были на клиенте), мы можем попытаться создать похожее. Если используем **бинарную сериализацию**, нам нужно четко знать, что мы собираемся восстановить. Причем **бинарная запись** – наиболее интересная. Она позволяет держать минимум информации, и поэтому в случае бинарной десериализаций мы должны четко понимать, что мы восстанавливаем. А в случае XML, JSON мы можем легко написать классы, которые будут выполнять десериализацию.

В .Net Framework существует ряд классов, которые позволяют автоматизировать процесс сериализации: 1) Первый класс скорее атрибут (хотя атрибуты – это класс) – атрибут **Serializable**. Как только мы пишем этот атрибут (над нашим классом) у нас появляется возможность – уже объекты нашего класса сериализировать и десериализировать автоматически. “Как это делается?” – В .Net Framework есть несколько классов поддерживающие разные формы десериализации и сериализации. Эти классы способны с помощью пространство имен **Reflection** получить информацию об объекте, взять все публичные свойства, эти публичные свойства сохранить в формате XML, JSON, бинарный формат и т.д. Ну и затем в случае десериализации они умеют вернуть наш объект в исходное состояние. В некоторых случаях этого может быть недостаточно, к примеру: 1) У нас может быть достаточно комплексный объект (большой); 2) Чтобы его создать (создать объект) может понадобиться какие-то дополнительные процедуры, а не просто там пустой конструктор. Поэтому существует возможность еще и этим процессом управлять, если нас не устраивает автоматическая сериализация/десериализация – это реализовать интерфейс **ISerializable**. Внутри этого интерфейса описан метод **GetObjectData**, и это единственный метод, которое нужно реализовать. Метод GetObjectData позволяет выполнить сериализацию объекта. В нем мы сами прописываем все свойства, все данные,

которые мы хотим сериализовать, и не важно публичные, или приватные, или protected. *ПРИЛОЖЕНИЕ 1:*

Создание типа, поддерживающего сериализацию

```
[Serializable]
public class ServiceConfiguration : ISerializable
{
  ...
  public ServiceConfiguration(
    SerializationInfo info, StreamingContext ctxt)
  {
    ...
  }

  public void GetObjectData(
    SerializationInfo info, StreamingContext context)
  {
    ...
  }
}
```

Мы сами можем реализовать интерфейс ISerializable, если нас не устраивает тот вариант сериализации, который позволяет задать атрибут **[Serializable]** по умолчанию.

“Как сериализация происходит?” – Берутся все public и сохраняются в XML. Мы можем сохранить полностью всю информацию, которая позволит нам восстановить объект этого класса в том состоянии в котором он был.

Также в C# есть классы, которые позволяют нам автоматически (если у нас реализован интерфейс ISerializable или объявлен атрибут [Serializable], или и то, и другое (на самом деле мы должны указывать атрибут даже если реализуем интерфейс)) выполнять сериализацию и десериализацию. Первый класс – **BinaryFormatter**. *ПРИЛОЖЕНИЕ 2:*

Сериализация в бинарном формате

- Сериализация

```
ServiceConfiguration config = ServiceConfiguration.Default;
IFormatter formatter = new BinaryFormatter();
FileStream buffer = File.Create("C:\\fourthcoffee\\config.txt");
formatter.Serialize(buffer, config);
buffer.Close();
```

- Десериализация

```
IFormatter formatter = new BinaryFormatter();
FileStream buffer = File.OpenRead("C:\\fourthcoffee\\config.txt");
ServiceConfiguration config
  = formatter.Deserialize(buffer) as ServiceConfiguration;
buffer.Close();
```

В *ПРИЛОЖЕНИЕ 2* Мы создаем объект этого класса и у этого объекта есть метод **Serialize**. Метод **Serialize** получает ссылку на поток, и получает ссылку на объект нашего класса. То есть мы совершенно спокойно выполняем сериализацию. Десериализация выполняется аналогичным образом, с помощью метода **Deserialize**. Опять же таки ссылка на поток, здесь возвращается объект нашего класса.

Сериализация и десериализация в формате **JSON**:

JSON

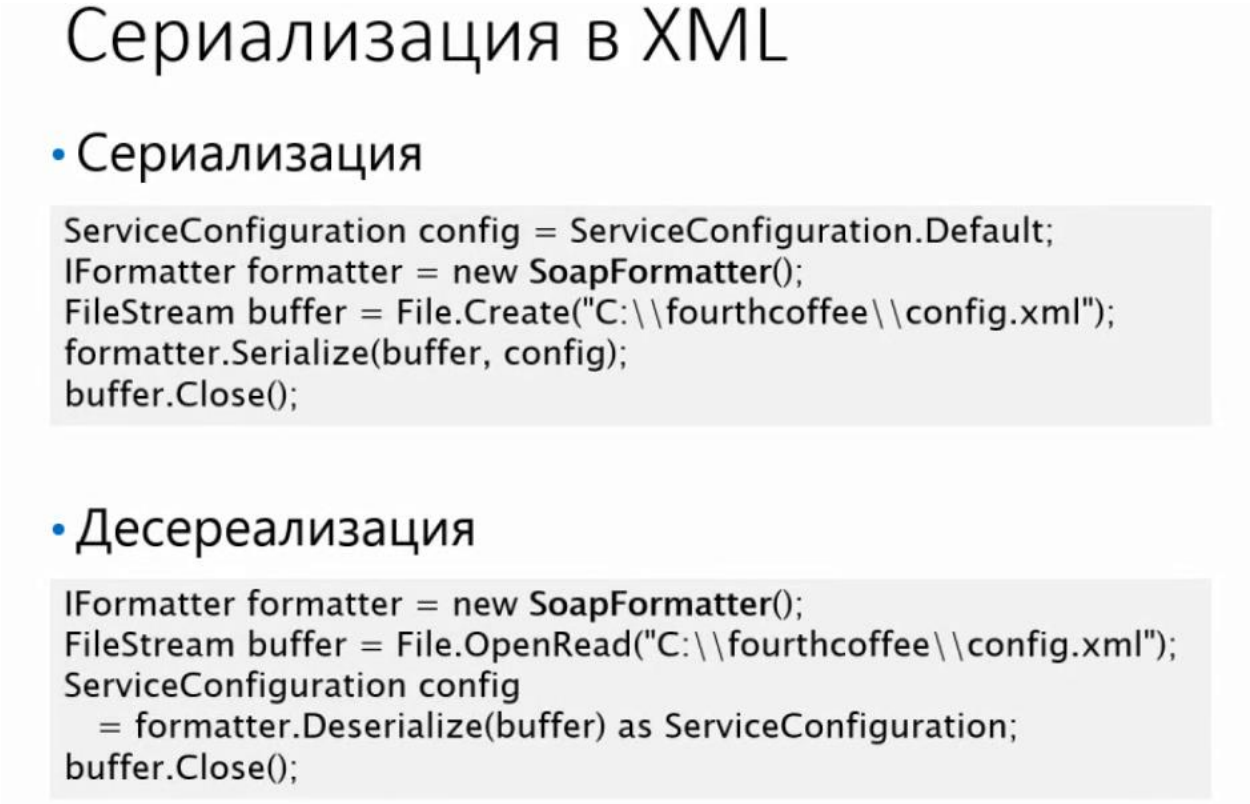
- Сериализация

```
ServiceConfiguration config = ServiceConfiguration.Default;
DataContractJsonSerializer jsonSerializer
  = new DataContractJsonSerializer(config.GetType());
FileStream buffer = File.Create("C:\\fourthcoffee\\config.txt");
jsonSerializer.WriteObject(buffer, config);
buffer.Close();
```

- Десериализация

```
DataContractJsonSerializer jsonSerializer = new
  DataContractJsonSerializer(
    typeof(ServiceConfiguration));
FileStream buffer = File.OpenRead("C:\\fourthcoffee\\config.txt");
ServiceConfiguration config = jsonSerializer.ReadObject(buffer)
  as ServiceConfiguration;
buffer.Close();
```

Сериализация и десериализация в **XML**:



Класс **SoapFormatter** позволяет выполнять сериализацию/десериализацию с помощью методов **Serialize** и **Deserialize**.

СОЗДАНИЕ СОБСТВЕННОГО СЕРИАЛАЙЗЕРА

Мы можем создавать собственный **сериалайзер**. Это мы будем делать, когда мы захотим сохранитьв каком-то собственном формате, то есть мы можем создавать свой собственный сериалайзер, реализовав интерфейс **IFormatter**, и соответственно здесь уже (в методах **Serialize**, **Deserialize**) выполнять сериализацию/десериализацию так как нам хочется.

RSS-файлы на сайтах – это **XML**.

При десериализации мы получаем XML (JSON, и т.д.) файлы, и преобразуем его в объекты. “Как это сделать?” – Есть способ, который базируется на атрибутах. В этом способе мы можем описать набор классов, которые будут четко соответствовать нашему XML-документу, прописав с помощью атрибутов какую часть класса мы на что хотим связать в нашем XML-документе. Пример:

ГЛАВА 13. РАБОТА С ТЕКСТОМ

В .Net Framework 1 символ занимает 16 бит (2 байта). Если мы хотим создать переменную типа char, то символ в отличий от строки задается с помощью одинарных кавычек. То есть, внутри одинарных кавычек мы можем написать любой символ. Есть некоторые символы, которые невидимы – они являются управляющими символами (переход на новую строку, и т.д.), а также есть символы, которые набрать с клавиатуры тяжело. Поэтому мы можем коды символов использовать для инициализации переменной типа char, то есть не просто символ в одинарных кавычках, а код символа. Для этого используется **“\”**, и после \ пишется код символа. char легко преобразовать к int даже без явного преобразования.

В .Net Framework все простейшие типы определены как структура. И это не мешает или быть пораженными (наследованы) от класса Object, и инкапсулировать все методы, которые есть в Object-е. Здесь и System.Char тоже не исключение, System.Char тоже является структурой.

Поскольку char – это структура, она содержит большое количество интересных методов и свойств. И один его из популярных методов это – метод **GetUnicodeCategory()**. И этот метод позволяет получить тип символа, потому что не всегда понятно с чем мы работаем, с каким конкретным типом. То есть, то ли этот символ (символ с которым мы работаем) является числом, то ли этот символ является буквой, то ли этот символ является прописной буквой, то ли заглавной буквой, то есть с помощью метода GetUnicodeCategory() мы можем получить эту информацию. Более того существуют ряд методов, которые по большему счету являются частной реализацией метода GetUnicodeCategory(), ну например, метод IsDigit() позволяет проверить “является ли значение внутри переменной типа char символом”. Переменная типа char является обычной двухбайтной переменной.

System.Char

- **Размер символа составляет 16 бит с минимальным значением ‘\0’ и максимальным ‘\uffff’**
- **GetUnicodeCategory() позволяет получить тип символа**
  - **Но можно использовать и более простые методы IsLower(), IsDigit(), и др.**
- **Преобразовать символы можно с помощью одного из методов ToLower(), ToLowerInvariant() и др.**
- **Легко преобразуется к int.**

Пример:

Пример 1

```
using System;
namespace Modul18
{
    class Program
    {
        static void Main(string[] args)
        {
            Console.WriteLine(char.GetUnicodeCategory('5'));
            Console.ReadLine();
        }}}
```

Результат: DecimalDigitNumber

Тип **String** является классом. Класс String позволяет задать строку. Строка задается с помощью двойных кавычек, и String – некий класс, который, во-первых, String является классом sealed, то есть от него нельзя наследоваться, во-вторых, String выделяется из всех типов за счет того, что .Net Framework знает как со строками работать “по-другому”, чем с остальными со всеми типами. “Что значит по-другому?” – Во-первых, строки в .Net Framework являются неизменяемыми, то есть когда мы создаем какую-то переменную типа string и инициализируем его, то ту строку которую мы создали для инициализации переменной уже нельзя изменить. Любая операция над этой строкой приводит появлению абсолютно новой строки. Это было сделано специально, чтобы с одной стороны облегчить работу со строками, потому что строки записывались в метаданные, с другой стороны понимая что строки могут быть различной длины и очень часто строки могут повторятся в .Net Framework была введена специальная структура – “**хэш-таблица**”. То есть, если переменные значения создаются в стеке, переменные каких-то классов (объекты классов) создаются в общей памяти, то для строк выделяется отдельная “хэш-таблица”. В этой хэш-таблице есть 2 параметра: 1) **Хэш-код** строки, которая присутствует у нас в приложении. 2) **Количество ссылок**, которые введут на ту или другую строку. То есть, идея состоит в том, что **GC** (**Garbage Collector**) дополнительно к общей памяти обрабатывает и хэш-таблицу. И соответственно, когда мы создаем две одинаковых строки, то у нас фактически память для двух строк не выделяется. У нас просто наращивается счетчик ссылок согласно тому сколько их у нас в приложении. “Как компилятор понимает, что это одинаковые строки?” – Он вычисляет хэш-код каждой строки, и по хэш-коду он понимает, что поскольку хэш-кода одинаковые, то и строки – одинаковые, и он берет и добавляет новый счетчик в хэш-таблицу, и не думает о том, что надо выделять новый кусок памяти. В классе String есть интересный метод – **ReferenceEquals()**. Он позволяет определить соответствуют ли, то есть являются ли две ссылки ссылками на один и тот же объект:

String.ReferenceEquals(имя_первой_переменной, имя_второй_переменной);

Результат этого метода, конечно же, true или false. То есть, если первая_переменная и вторая переменная в методе ReferenceEquals(), указывают на один и тот же объект в памяти, он возвращает true. Пример:

Пример 2

```
using System;
namespace Modul18
{
    class Program
    {
        static void Main(string[] args)
        {
            string s = "Hello";
            string s1 = "Hello";

            Console.WriteLine(String.ReferenceEquals(s, s1));
            Console.ReadLine();
        }}
}
```

Результат: True

Сейчас в компиляторах есть флажок, который позволяет избежать ссылку на один объект, который позволяет заставить выделять память для каждой строчки в общей памяти.

В классе String есть метод – **Intern()**, который позволяет гарантировать создание строки в хэш-таблице. То есть, если у нас существуют в приложении много строк, и эти строчки мы хотим между собой сравнивать, то как происходит сравнение строк? – По каждому символу – это очень долго. Если мы предполагаем, что у нас много похожих строк есть, то есть одинаковых, то нам лучше гарантировать их сохранение в хэш-таблицы, тогда мы можем выполнять сравнение не по символьно, а с помощью метода ReferenceEquals(). И подобное сравнение будет выполняться очень быстро. Базируясь на этом мы можем просто выполнять любые операции сравнение: равны ссылки значит строки одинаковые, неравны ссылки значит строки разные.

Строку модифицировать (изменить первоначальное значение) нельзя, то есть строка неизменяемая. Любые методы, которые работают со строками генерируют новую строку и возвращают нам, то есть будет добавлена в хэш-таблицу, но никак не изменяют исходную строку. В следующем примере мы покажем пример изменение строки **s1**. Пример:

Пример 3

```
using System;
namespace Modul18
{
    class Program
    {
        static void Main(string[] args)
        {
            string s = "Hello";
            string s1 = "Hello";

              /* Здесь мы изменяем строку Hello, и опять должны присвоить s1. Потому
что при изменении строки Hello метод Replace создает и возвращает новую переменную и мы
должны ее присвоить опять к s1 */
            s1 = s1.Replace('e','b');

            Console.WriteLine(s1);
            Console.ReadLine();
        }}}

Результат: Hbllo
```

Если мы хотим чтобы написанные в строке специальные символы игнорировались, то есть стали как обычные символы мы перед строкой прописываем специальный символ **@**. Пример:

Пример 4

```
string s1 = @“He\nllo”;
```

В *Примере 4* специальный символ “\n” игнорирует и читается как обычные символы “\” и “n”. Любая операция со строками приводит к генерации новых строк. Операции (методы) внутри строки достаточно много, это – возможность замены символов, удаление, преобразование их в другие форматы, клонирование, сравнение – все это присутствует внутри класса String, и соответственно доступно через объект класса или через сам класс String. Но всегда нужно помнить, что строки неизменяемыми. То есть, в принципе если мы планируем делать большое количество операции со строками, не просто их сравнивать, а

модифицировать, то использовать класс String не оптимально, потому что при каждой его модификации будут создаваться новые переменные (строки), и будет увеличивать нашу хэш-таблицу, заставляет GC работать более активно, и это нехорошо. Альтернативным решением могло бы быть “создание динамической структуры”, которая бы позволило бы работать со строками в памяти. То есть, по большему счету динамическая структура типа char (некий список из символов, который бы мог интерпретироваться как строка, и который бы эффективно позволил бы работать с этой строкой в памяти). К примеру, список (List) типа char. Нам не нужно ничего создавать, такая структура уже есть, это – **StringBuilder**. Он находиться в пространстве имен **System.Text.StringBuilder** – это запечатанный класс. Класс StringBuilder позволяет работать со строками как с неким динамическим массивом типа char. И всякий раз если мы планируем большое количество операции со строками вместо String используйте StringBuilder. Тогда мы гарантируем, что строки создаются в памяти и мы неэффектим производительность. Естественно, любая операция с каким-то экземпляром типа StringBuilder будет эффектить, именно этот StringBuilder. И понятно, что в какой-то момент времени мы можем прекратить работу с StringBuilder-ом и преобразовать его содержимое к строке. Пример создания объекта класса StringBuilder:

Пример 5

StringBuilder b = new StringBuilder();

В классе StringBuilder много разных конструкторов, и поэтому мы можем передавать в качестве параметра объекту класса StringBuilder разное количество и типы параметров. Здесь можно задать количество возможных символов, также можно задать строку в качестве параметров. Пример:

Пример 6

StringBuilder b = new StringBuilder(“Hello”);

В классе StringBuilder также имеется большое количество методов. И когда мы используем класс StringBuilder, у нас не создается новая строка как в String-е и нам не нужно будет его куда-то присваивать и т.д., а мы можем работать (модифицировать) с этой строкой в памяти.

То есть, любые операции со StringBuilder-ом приводит к изменению этого же объекта класса StringBuilder.

Строки можно форматировать и есть варианты, связанные с преобразованием каких-то готовых классов к строкам. Рассмотрим несколько методов: 1) Для переменной типа int – **Parse()** позволяет взять строку и преобразовать ее в переменную типа int. Если у нас в строке будет несовместимые символы (к примеру, буквы), то метод Parse() будет выкидывать исключение. Если метод **TryParse()**, который может проверить возможность преобразование, и он возвращает булево значение. Метод Parse() и метод TryParse() также имеются в типах double, float, char, и в этих типах она тоже позволяет преобразовать строку в соответствующий тип, и проверить можно ли преобразовать строку (TryParse()) в соответствующий тип:

System.Text.StringBuilder

- **Более эффективный способ работы со строками;**

- **Любые операции возвращают ссылку на тот же объект.**

БЕЗОПАСНЫЕ СТРОКИ – **System.Security.SecureString**

Строки в .Net Framework не защищены, и мы не можем контролировать механизм их удаления. То есть, если нету и одной ссылки, это не означает, что строка из памяти была удалена, GC мог еще по ней пробежаться. А с другой стороны, даже если GC вроде бы очистил память, он не всегда обязан что-то обнулить, особенно, при работе с хэш-таблицами эта строка может где-то в памяти остаться до того как ее перепишет что-то еще. Чтобы контролировать процесс хранения данных (строк), то есть четко знать когда они будут уничтожены, и в случае уничтожении их с памяти можно вытереть с помощью класса **SecureString**. Особенность класса SecureString состоит в том, что он позволяет создать строку в неуправляемой памяти, то есть в памяти, который не будет управляться GC-ом. И создавая подобную строку в этой памяти мы фактически возлагаем полностью на себя контроль по уничтожению этого объекта. С одной стороны, мы возлагаем на себя дополнительные обязанности, с другой стороны, мы можем начиная с этого момента контролировать процесс уничтожения строк в этой неуправляемой памяти, для этого есть метод **Dispose()**.

Если нам понадобится работать со строками в неуправляемой памяти, то это можно сделать с помощью класса SecureString.

ГЛАВА 14. КОЛЛЕКЦИИ

Те структуры данных которые у нас есть – массивы, элементарные типы не всегда удовлетворяет наши потребности. К примеру, с массивами основная проблема состоит в том, что они имеют фиксированную длину, то есть мы должны при создании массива точно знать сколько там будет элементов и выделить необходимое количество памяти, это во многих случаях бывает плохо, потому что нам приходится либо выделять больше памяти чем нужно, либо же пытаться каждый переписывать(создавать) новые массивы если хотим внести какие-то изменения. То есть то, что массив нединамический очень неудобно.

ЧТО НЕ ТАК С МАССИВАМИ?

- Массивы имеют фиксированную длину(мы должны знать количество элементов в момент создания массива)
- Некоторые алгоритмы плохо работают для массивов
  - Проблемы с перестановкой и вставкой элементов
  - Сортировка, поиск
- Хранение однотипных элементов

В C# так называемые “коллекции старого типа”, которые тянутся с первой версии .Net Framework. Все эти коллекции(ArrayList, Hashtable, Queue, Stack) можно найти в двух пространствах имен – System.Collections и System.Collections.Specialized. Эти коллекции(ArrayList, Hashtable, Queue, Stack) являются динамическими.

ArrayList – список, который позволяет легко работать с элементами в любой части динамической структуры.

Hashtable – хэш-таблица.

Queue – очередь.

Stack – стек.

В пространствах имен System.Collections и System.Collections.Specialized коллекции значительно больше. А ArrayList, Hashtable, Queue, Stack – это наиболее популярные.

Array – это класс, который описывает статические методы, которые применимы к массиву. И Array по большему счету это-некая абстракция, которая используется для создание массива. Это – вообще нединамическая коллекция. Также мы не можем создавать объекты класса. А вот объекты классов ArrayList, Hashtable, Queue, Stack мы можем создавать, начинать добавлять элементы, и начинать с этими элементами работать. “Почему эти коллекции старого типа и их начали редко использовать сейчас?” – Когда создавались эти коллекции не было понятие обобщенного типа, и поэтому все эти коллекции(ArrayList, Hashtable, Queue, Stack) работают с элементами типа object. То есть, они с одной стороны небезопасные коллекции, то есть это значит что туда можно положить все что угодно, то есть они нетипизированы. Когда мы туда ложим разные типы при работе с ними возникает проблемы, потому что к разным типам разные операции и т.д., и еще мы должны объекты этих коллекции каждый раз явно преобразовывать в нужный нам тип с типа object, потому что все объекты этой коллекции хранятся типом object. Пример:

Пример 1

```
using System;
using System.Collections;
```

```
namespace Modul19
{
    class Program
    {
        static void Main(string[] args)
        {
            ArrayList list = new ArrayList();
            list.Add(3);

            foreach (object ob in list)
            {
                Console.WriteLine((int)ob);
            }}}}
Результат: 3
```

Пример 2

```
using System;
using System.Collections;

namespace Modul19
{
    class Program
    {
        static void Main(string[] args)
        {
            ArrayList list = new ArrayList();
            list.Add('5');

            foreach (object ob in list)
            {
                Console.WriteLine((char)ob);
            }}}}
Результат: 5
```

КОЛЛЕКЦИИ СТАРОГО ТИПА

- Пространства имен System.Collections и System.Collections.Specialized
- Основные типы коллекции
  - ArrayList
  - Hashtable
  - Queue
  - Stack
- Небезопасные коллекции
- Имеют проблемы с производительностью

Во втором C# проблема была в принципе решена. “Каким образом она была решена?” – ввели понятие обобщенных типов. Все коллекции, которые используют обобщенные типы(Generics) находятся в пространство имен System.Collections.Generics. С помощью обобщенных типов мы можем коллекции строго типизировать. То есть, мы создавая объект коллекции указываем тип элемента, который будет храниться внутри коллекции(то ли это простейший тип, то ли это тип нашего класса, и т.д.). Далее при работе с этой коллекцией мы совершенно спокойно работаем с коллекцией зная что мы туда можем

добавить только элементы указанного типа. Наиболее популярная коллекция это – **List<T>**. Вторая по популярности коллекция это – **Dictionary<TKey,TValue>**. Dictionary<TKey,TValue> позволяет хранить ключ и значение. Dictionary<TKey,TValue> параметризируется двумя типами(типом ключа TKey и типом значение TValue), поскольку ключ у нас может быть произвольного типа(ключ может быть числового типа, может быть хэш-кодом, строкой и т.д.), ну и значение тоже может быть объектом любого класса. Dictionary<TKey,TValue> параметризируется двумя типами, а коллекции – Stack<T>, Queue<T>, List<T> параметризируется одним типом. В этом примере мы опишем List, который хранит переменные типа int:

```
List<int> list = new List<int>();
```

Пример:

Пример 3

```
using System;
using System.Collections;
using System.Collections.Generic;

namespace Modul19
{
    class Program
    {
        static void Main(string[] args)
        {
            List<int> list = new List<int>();
            list.Add(3);
            list.Add(5);
            foreach (int r in list)
            {
                Console.WriteLine(r);
            }
            Console.ReadLine();
        }}
}
```

Результат: 3 5

КОЛЛЕКЦИИ НОВОГО ТИПА

- Используют обобщенные типы
  - List<T> - обычный список
  - LinkedList<T> - каждый элемент этого списка связан с предыдущим и следующим
  - Stack<T> - стек
  - Queue<T> - очередь
- Не имеют проблем старых коллекции

Выше сказанные коллекции являются из .Net Framework. Если говорить про Windows 8, то в Windows 8 есть еще одна коллекция – **ObservableCollection<T>**. Она

находится в пространстве имен **System.Collections.ObjectModel**. Эта коллекция нужна для приложении Windows 8, а также для приложении Windows Presentation Foundation, Silverlight, Windows Phone. ObservableCollection<T> - тип коллекции, которая может нотифицировать(давать знать) графический интерфейс, в случае его связки с коллекцией о том, что внутри коллекции произошли какие-то изменения, и интерфейс будет автоматический обновляться. Эта коллекция используется преимущественно в разработке под Windows Phone и Windows 8. По большему счету, это тот же самый List, но с реализацией, которая может нотифицировать графический интерфейс о каких-то изменениях внутри коллекции. Это удобно потому что связывание(создание) графического интерфейса и коллекции идет только один раз, то есть мы связали один раз графический интерфейс со своей коллекцией – данные в графическом интерфейсе появились, и графический интерфейс затем не следит за тем, что случается внутри коллекции. А ObservableCollection дает такой механизм, и тогда графический интерфейс, может следить за тем какие изменения произошли в коллекции, и как только изменения произойдут графический интерфейс обновляет данные.

КОЛЛЕКЦИИ НОВОГО ТИПА

- Словари, позволяющие хранить пары в виде ключ/значение:
  - Dictionary<TKey,TValue>
  - Sorted Dictionary<TKey,TValue>

Если мы запускаем несколько потоков, и во всех потоках мы получаем доступ к одной и той же коллекции, причем в одной пытаемся что-то добавить, в другой пытаемся что-то удалить, это может привести к проблеме, потому что у нас доступ из разных потоков идет конкурентной и нами выше рассмотренные коллекции ни коем образом это не контролируют. Поэтому если мы знаем, что у нас доступ к коллекциям будет из разных потоков, то здесь рекомендуется использовать специальные коллекции для этого, такие как **ConcurrentStack**, **ConcurrentDictionary**, **ConcurrentQueue**. Они тоже являются Generics-ами, то есть поддерживают обобщенные типы. В тех случаях, если идет работа с коллекцией сразу в нескольких потоках рекомендуется использовать эти коллекции: **ConcurrentStack**, **ConcurrentDictionary**, **ConcurrentQueue**. Они находятся в пространстве имен: **System.Collections.Concurrent**.

Коллекции типа Stack<T>, Queue<T>, List<T> наследуются от интерфейса **IList<T>**.

А коллекции, которые работают с двумя значениями(ключ, значение) наследуются от **IDictionary<TKey,TValue>**, точнее реализуют этот интерфейс.

Если нам понадобится сделать собственную коллекцию, то понятно что лучше базироваться на стандартных интерфейсах, чтобы сделать так, чтобы эта коллекция подчинялась всем тем принципам, которые идут из .Net Framework. Основным интерфейсам, который обязательно надо придумать(реализовать) – это интерфейс **IEnumerable**. IEnumerable позволяет сделать коллекцию перечисляемой. “Что значит перечисляемой?” – Это значит реализовать внутри нее механизм, который позволит использовать цикл foreach, то есть пробежаться по объектам коллекции. Пример:

Пример 4

```
using System;
using System.Collections;
using System.Collections.Generic;
namespace Modul19
```

```
{
    class Program
    {
        static void Main(string[] args)
        {
            List<int> list = new List<int>();
            list.Add(3);
            list.Add(5);
            foreach (int r in list)
            {
                Console.WriteLine(r);
            }

            Console.ReadLine();
        }
}

    // Создаем свой собственный List, которого назовем MyList
    class MyList<T>: IEnumerable<T>
    {
        // Переменная,которая будет ссылатся на root-овый Node
        Node<T> root;

        class Node<T>
        {
            public T value;
            // Ссылка на следующий Node
            public Node<T> next;
        }

        // Нам нужно создать метод Add, который добавляет очередной Node в наш список
        public void Add(T newValue)
        {
            // В нашем списке мы будем добавлять Node в голову нашего списка.
            /* Здесь мы проверяем root является на null, то есть мы проверяем нету ли у
нас элементов в списке */
            if (root == null)
            {
                // Создаем новый элемента Node-а
                root = new Node<T>();
                // И в root.value сохраняем наш newValue
                root.value = newValue;
            }
            else
            {
                // В противном случае мы создаем новую Node-у
                Node<T> newNode = new Node<T>();

                /* И в этой новой Node-е мы обьязательно инициализируем value в значение
newValue */
                newNode.value = newValue;

                // У нас next в новой Node-е должен указывать на root
                newNode.next = root;

                // И root надо переместить
                root = newNode;
            }}
        /* Выше написанном коде мы создали новую Node-у. Новую Node-у связали с
головой, и голову переместили вперед. То есть здесь мы добавляем в голову новый
элемент. */
    }
}
```

Если мы хотим реализовать в нашем собственном коллекции цикл foreach, то есть если нам нужно выбрать все элементы, то есть если мы хотим механизм, который последовательно эти элементы возвращает. Для этого нам надо реализовать интерфейс IEnumerable<T>. В интерфейсе IEnumerable<T> есть два метода:

1) **GetEnumerator()**

2) **IEnumerable.GetEnumerator()**

Метод GetEnumerator() возвращает все элементы последовательно. Пример:

Пример 5

```
using System;
using System.Collections;
using System.Collections.Generic;

namespace Modul19
{
    class Program
    {
        static void Main(string[] args)
        {
            MyList<int> list = new MyList<int>();
            list.Add(3);
            list.Add(5);
            foreach (int r in list)
            {
                Console.WriteLine(r);
            }

            Console.ReadLine();
        }
    }

    // Создаем свой собственный List, которого назовем MyList
    class MyList<T>: IEnumerable<T>
    {
        // Переменная,которая будет ссылатся на root-овый Node
        Node<T> root;

        class Node<T>
        {
            public T value;
            // Ссылка на следующий Node
            public Node<T> next;
        }

        // Нам нужно создать метод Add, который добавляет очередной Node в наш список
        public void Add(T newValue)
        {
            // В нашем списке мы будем добавлять Node в голову нашего списка

            /* Здесь мы проверяем root является на null, то есть мы проверяем нету ли у
нас элементов в списке */
            if (root == null)
            {
                // Создаем новый элемента Node-а
                root = new Node<T>();
                // И в root.value сохраняем наш newValue
                root.value = newValue;
            }
            else
            {
```

```
                // В противном случае мы создаем новую Node-у
                Node<T> newNode = new Node<T>();

                /* И в этой новой Node-е мы обьязательно инициализируем value в значение
newValue */
                newNode.value = newValue;

                // У нас next в новой Node-е должен указывать на root
                newNode.next = root;

                // И root надо переместить
                root = newNode;
            }
        }
        /* Выше написанном коде мы создали новую Node-у. Новую Node-у связали с
головой, и голову переместили вперед. То есть здесь мы добавляем в голову новый элемент
*/

        // Метод GetEnumerator() должен возвращать все наши элементы
        public IEnumerator<T> GetEnumerator()
        {
            // Здесь мы создадим временную Node-у и получим ссылку на root-овую Node-у
            Node<T> tempNode = root;

            // Раз мы хотим вернуть все элементы, нам нужен цикл while
            while (tempNode != null) /* Этот цикл будет проходить пока мы не уткнемся в
конец списка */
            {
                /* Конструкция yield return позволяет вернуть значение. Цикл foreach
обращается именно к методу yield, этот метод возвращает значение и ждет, цикл foreach
делает какие-то действия и ко второму, возвращает второе значение и ждет, и т.д.
Побольшему счету, yield return реализует возврат целого списка значений */
                yield return tempNode.value;
                /* То есть конструкция yield return будет возвращать значение до тех
пор пока while работает. Как только мы выходим из метода, foreach прекращает работу, то
есть он видит, что метод закончился и там значений больше нет.

                /* С помощью этого цикла мы будем проходится от головы до конца до тех
пор пока мы не уткнемся в конец списка. В этом цикле мы должны возвращать все элементы
(return tempNode.value) */
                tempNode = tempNode.next;
                /* То есть мы взяли стали в начале списка, и бежим от головы списка до
его конца до тех пор пока не упремся в конец */
            }
        }

        IEnumerator IEnumerable.GetEnumerator()
        {
            throw new NotImplementedException();
        }

    }
}
```

Результат: 5 3

В цикле никогда нельзя менять динамическую структуру по которому мы бежим. “Что значит менять?” – Добавляем туда элементы, удалять оттуда элементы и т.д. Это всегда приведет к сбою цикла foreach. То есть, бежит цикл foreach по элементам и тут исчезают или начинают появляться новые элементы в списке, и не понятно как ему в этом случае реагировать.

И за счет **yield return** в GetEnumerator() мы можем и не реализовывать метод **IEnumerable.GetEnumerator()**.

РАБОТА С ПЕРЕЧИСЛИМЫМИ ТИПАМИ

- Требуется реализовать интерфейс IEnumerable
- Реализовать GetEnumerator метод
- Испоьзуйте yield return для реализации итератора

Также мы можем в нашем классе(коллекций) реализовать индексатора, метод, Remove(), свойства, и т.д.

ГЛАВА 15. ИСПОЛЬЗОВАНИЕ LINQ

Linq представляет собой своеобразный язык запросов, который позволяет выбирать данные из различных источников таких как – наборы данных в памяти, XML-документы, базы данных и т.д.

Чтобы инициализировать public данные класса сразу при создании его объекта, то есть не писав такую конструкцию: **имя_объекта.публичное_данное = значение;** мы должны прописать в фигурных скобках после объекта класса: **данное = значение**. Пример:

Пример 1

```
using System.Collections.Generic;
namespace Modul20
{
    class Program
    {
        static void Main(string[] args)
        {
            List<Person> p = new List<Person>();
            p.Add(new Person() { FirstName = "Sergey", LastName = "Baydachnyy", Age =
35, Address = "Kirova 121"});
        }}

    class Person
    {
        public string FirstName { get; set; }
        public string LastName { get; set; }
        public int Age { get; set; }
        public string Address { get; set; }
    }
}
```

Инициализация объектом таким образом (*Пример 1*) выглядит очень просто. Все свойства (переменные), которых мы инициализируем таким образом, мы должны разделять через запятые (как показано в *Примере 1*). Таким образом, мы можем инициализировать сколько угодно много свойств (переменные), можем выбрать какие нужны, то есть можем не всех их инициализировать. Затем инициализировав все нужные свойства (переменные) мы закрываем эту конструкцию фигурными скобочками (как показано в *Примере 1*).

Ключевое слово **var** позволяет определять переменную инициализировать ее не указывая явно тип при определений переменной. К примеру:

Пример 2

1) var p = new List<Person>();
2) var a = 5;

При определение переменной типа var, она должна сразу же быть инициализирована.

В C# есть возможность создавать **анонимные типы**. “Что это значит?” – Это значит, что мы можем создать объект некоторого типа (класса), который у нас неопределен как класс. То есть, он будет сформирован фактически налету. “Как это делается?” – Мы используем ключевое слово **new**, а за ключевым слово new в фигурных скобках, мы и создаем анонимный тип. При создании анонимного типа мы используем тот же синтаксис, что и при инициализации свойств при явном создании типов (как показано в *Примере 1*). То есть, мы в фигурных скобках придумываем и инициализируем абсолютно новые

свойства. Тип каждого свойства выводится компилятором (то есть, по типу значения, которым его инициализируем). Пример:

Пример 3

var p = new{Name = “Sergey”, Street = “Kirova”};

После оператора new не пишется имя типа (класса): **анонимные типы позволяют легко инкапсулировать свойства только для чтения в один объект без необходимости предварительного определения типа**. После оператора new в фигурных скобках проводим инициализацию тех свойств, которых мы считаем нужными в нашем новом типе. “Что в этот момент произойдет?” – В этот момент произойдет создание нового типа (нового класса), имеющего свойства, которых мы прописали в фигурных скобках, и тут же будет создан объект этого класса, где будут инициализированы свойства, которых мы прописали в фигурных скобках. Анонимные типы содержат один или несколько публичных свойств только для чтения. Другие члены класса, например методы или события недопустимы. Здесь ключевое слово var и помогает в определении переменных анонимного типа, потому что мы не знаем тип анонимного типа. Пример:

Пример 4

var p = new{Name = “Alik”, Age = 5};

То есть, мы возлагаем на компилятор определение типа переменных, типа “анонимного типа”, то есть компилятор создает класс самостоятельно, и подставляет имя_класса вместо ключевого слово var. Вот поэтому и важен var в этой конструкций. *Примечание*:

VAR И АНОНИМНЫЕ ТИПЫ

- **var – позволяет определять переменную, давая возможность компилятору определить тип самостоятельно;**
- **переменная типа var должна быть инициализирована в момент объявления;**
- **анонимный тип – облегченный механизм создания контейнера для наших данных.**

ИНИЦИАЛИЗАЦИЯ ОБЪЕКТА

Начиная с C# 3.0 появилась возможность инициализировать свойства объекта (public) при создании объекта.

РАСШИРЕНИЕ

Еще одна функциональность C# – **расширение**. “Что такое расширение?” – У нас есть ряд классов, это могут быть наши классы, это могут быть классы .Net Framework. Но если мы видим, что эти классы не обладают нужными нам методами, и мы хотим расширить возможности этих классов, фактически добавив туда некоторые свои методы.

Предположим есть класс String (стандартный класс .Net), и нам нужен метод, который поможет добавить нужный нам метод (наш метод) в класс String. Методы расширения с точки зрения разработчика выглядят как методы встроенные в существующие уже классы.

Статический класс, статический метод и ключевое слово this перед определением параметра типа класса, которую мы собираемся расширять позволяет нам интегрировать наш метод внутрь класса, которую мы собираемся расширять. Фактически хоть это и внешний метод, но для разработчика он будет выглядеть как внутренний метод расширяемого класса.

Вот такая вот конструкция: **“статический класс, статический метод и ключевое слово this перед параметром, которая определяется классом, которую мы с вами расширяем”** позволяет нам интегрировать наши методы внутрь нашего класса. Причем параметр на которую мы указываем ключевым словом this и который определяет класс (тип) в которую мы хотим интегрировать новые методы (расширять) при вызове метода не воспринимаются как параметр (соответственно, мы не передаем ему аргументы), этот параметр будет переменной у которого мы вызовем наш расширяемый метод. Пример:

Пример 5

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace Modul20
{
   class Program
   {
      static void Main(string[] args)
      {
         string y = "Hello";

         /* При вызове метода RemoveSpaces() мы не передаем никакие аргументы, потому что переменная s не воспринимается как параметр, а будет как переменная у которого мы вызываем метод RemoveSpaces */
         Console.WriteLine(y.RemoveSpaces());

         Console.ReadLine();
      }}
   // Напишем код для расширение класса String. Имя_класса может быть любым
   static class StringExtender
   {
      /*  Метод, который будет расширять класс String. Какой класс(тип) мы будем расширять мы указываем типом параметра указав на него ключевым словом: this. В нашем примере класс string */
      static public string RemoveSpaces(this string s)
      {
         return "Modified string";
      }
}
}
```

Результат: Modified string

Пример 6

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace Modul20
{
   class Program
   {
      static void Main(string[] args)
      {
        int y = 5;

         /*  При вызове метода RemoveSpaces() мы не передаем никакие аргументы, потому что переменная s не воспринимается как параметр, а будет как переменная у которого мы вызываем метод RemoveSpaces */
         Console.WriteLine(y.RemoveSpaces());

         Console.ReadLine();
      }}
   // Напишем код для расширение класса Integer. Имя_класса может быть любым
   static class IntegerExtender
   {
      /* Метод, который будет расширять класс Integer. Какой класс(тип) мы будем расширять мы указываем типом параметра указав на него ключевым словом: this. В нашем примере класс int */
      static public string RemoveSpaces(this int s)
      {
         return "Modified string";
      }
}
}
```

Результат: Modified string

Понятно, что метод не встраивается в класс, которую мы хотим расширить, а он лежит где-то отдельно, но при этом компилятор их связывает. При этом надо помнить, что класс, которую мы хотим расширить и класс, который содержит расширяемый метод должны лежать в одном пространстве имен:

РАСШИРЕНИЯ

- **Позволяет расширить функциональность существующих классов;**
- **Определяются в виде статических классов;**
- **Описывается статическими методами;**
- **Должны быть описаны в “правильном” пространстве имен;**

- **Определяются по параметру (с ключевым словом).**

LINQ

Linq (Language Integrated Query) – методология от Microsoft, которая расширяет возможности C# позволяя создавать запросы к данным.

Language Integrated Query – язык интегрированных запросов. То есть, Linq – язык запросов, которая представляет собой команды на языке программирования C#, который позволяет создавать запросы к различным источникам данных. Основные источники данных:

1) **.Net Framework коллекции**, то есть все вещи (коллекции), которые реализуют интерфейс **IEnumerable**, то есть все коллекции. Мы можем обращаться к коллекциям используя язык LINQ.
2) **SQL Server**. Взаимодействие с Sql Server-ом, то есть с данными, которые выбираются с Sql Server-а.
3) **ADO.NET**.
4) **XML**-документы. То есть, с XML документами тоже можно с помощью LINQ работать.

LINQ позволяет выбирать данные, и фильтровать данные. То есть как и во всех стандартных языках запросов LINQ позволяет ставить условие, фильтровать данные и т.д. Расширения можно использовать не только по отношению к классам, но и к интерфейсам. Методами LINQ-а был расширен интерфейс **IEnumerable<T>**, который реализуется всеми коллекциями и следовательно расширения, которые позволяют нам обращаться к данным в этих коллекциях появились сразу же автоматически во всех коллекциях. LINQ – некий сторонний компонент .Net Framework-а, то есть он не тесно интегрируется поскольку это отдельный набор классов с коллекциями. Но за счет того, что у нас есть возможность писать расширители, то вообщем-то LINQ хорошо влился в существующую инфраструктуру. Основные методы, которые используются в LINQ:

1) **Select** – позволяет выбирать данные.
2) **Where** – позволяет строить фильтры.
3) **OrderBy** – позволяет выполнять сортировку.

LINQ представляет из себя на физическом уровне библиотеку с реализацией большого количество методов-расширителей.

ОПЕРАТОРЫ В LINQ

- **Реализуются с помощью методов расширений в C#.**
- **Расширяют IEnumerable<T>, а следовательно доступны в коллекциях.**
- **Примеры методов: Select, Where, OrderBy.**

ЛЯМБДА ВЫРАЖЕНИЯ

Лямбда выражение – упрощенный синтаксис для создания анонимных методов.

Мы уже рассматривали “анонимные типы”, а здесь мы рассмотрим понятие “**анонимные методы**”. **Анонимные методы** – это некая упрощенная запись методов с помощью оператора **=>**. С левой стороны этого оператора (**…**=>) пишутся параметры,

которые фактически тоже генерируются (создаются) на лету, которые выбираются из коллекции и передаются в анонимные методы. А с правой стороны этого оператора (=>**…**) пишется тело анонимного метода, причем в зависимости от размера этого это может быть одна запись или это может быть там целый набор операторов. Пример:

Пример 7

```
p.Where (item => item.Age <= 30);
```

p – коллекция.
item – параметр, который будет хранить элемент коллекции.

Метод Where() позволяет перебрать элементы коллекции под какое-то условие (в нашем примере item.Age <= 30). То есть, с помощью ключевого слово Where, мы можем настроить фильтр. Для того, чтобы создать коллекцию мы должны использовать метод Select(). Метод Select() берет в качестве своего параметра берет каждый элемент, который соответствует нашему критерию (в блоке Where(…)).

Метод Where() позволяет перебрать элементы под какое-то условие (в нашем примере item.Age <= 30), и все выбранные элементы можно сохранять в какой-то параметр (в нашем примере мы назвали этот параметр как item): **p.Where(item => item.Age <= 30);**

И с помощью оператора (=>) мы пишем, что создаем анонимное выражение, то есть мы будем передавать в переменную item наш новый очередной элемент, который соответствует условию справа от оператора =>. То есть, с помощью метода Where() мы можем настроить фильтр, но этого недостаточно. Здесь мы получим только переменную (item), которую использовать напрямую нельзя. Для того, чтобы создать явную коллекцию, то есть для того, чтобы сгенерировать коллекцию, мы должны использовать метод Select(). Метод Select() в качестве параметра принимает выбранный нами элемент (**it**) на основе наших условий с помощью Where():

```
…Select(it => it);
```

И дальше мы пишем, что мы хотим выбрать (***it***) (как мы хотим выбрать). То есть, мы по большему счету берем каждый “выбранный нами с помощью метода Where() элемент (**it**)”, выбираем этот же самый элемент it (и **it**, и ***it***). И соответственно будем получать некую коллекцию. Еще мы можем с помощью анонимных типов создать абсолютно новую коллекцию (коллекцию нового типа) вместо элемента it, который написан справа от оператора =>. Пример:

Пример 8

```
…Select(it => new{Name = String.Format(“{0}{1}”, it.FirstName, it.LastName), Age = it.Age});
```

При создании коллекции таким образом мы не будем знать их тип, и здесь нас выручает ключевое слово **var**. То есть, типом нашей новой коллекции мы обозначаем var, и компилятор сам подберет тип для нашей новой коллекции:

Пример 9

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace Modul20
{
    class Program
    {
        static void Main(string[] args)
        {
            List<Person> p = new List<Person>();
            p.Add(new Person() { FirstName = "Sergey", LastName = "Baydachnyy", Age =
35, Address = "Kirova 121"});
            p.Add(new Person() { FirstName = "Viktor", LastName = "Baydachnyy", Age =
34, Address = "Kirova 121" });
            p.Add(new Person() { FirstName = "Sergey", LastName = "Poplavskiy", Age =
30, Address = "Lenina 121" });
            p.Add(new Person() { FirstName = "Margarita", LastName = "Ostapchuk", Age =
15, Address = "Bandera 13" });

            /* Метод Where() позволяет перебрать элементы коллекции под какие-то
условие. В параметр item мы можем сохранить все отфильтрованные элементы коллекции, то
есть элементы, которые соответствуют условию - item.Age <= 30. Метод Select() берет в
качестве параметра каждый элемент, который соответствует нашему критерию в блоке
Where(...) слева от нашего оператора =>, в нашем примере элемент "it" */
            var u = p.Where(item => item.Age <= 30).Select(it =>  new { Name =
String.Format("{0} {1}", it.FirstName, it.LastName), Age = it.Age });
            Console.WriteLine(u.Count());

            Console.ReadLine();
        }
    }

    class Person
    {
        public string FirstName { get; set; }
        public string LastName { get; set; }
        public int Age { get; set; }
        public string Address { get; set; }
    }
}
```

Результат: 2

Причем созданная нами “выборка” не имеет определенного типа коллекции, которых мы знаем (List, Dictionary, etc.). Созданная нами “выборка” – коллекция, которая реализует интерфейс IEnumerable. И мы дальше можем продолжать с этой коллекцией работать используя методы что и у других коллекции. Если мы хотим дальше работать с этой коллекцией как с List, Dictionary, Array (массив) и т.д., то мы должны предварительно вызвать один из методов приведения:

1) ToArray()

2) ToDictionary()

3) ToList() и др.

И после приведения нашей коллекций к определенному типу коллекций, нам будут доступны его свойства и методы. А до того как мы преобразуем его в определенный тип, то есть в исходном варианте у нас коллекция будет типа IEnumerable. Пример:

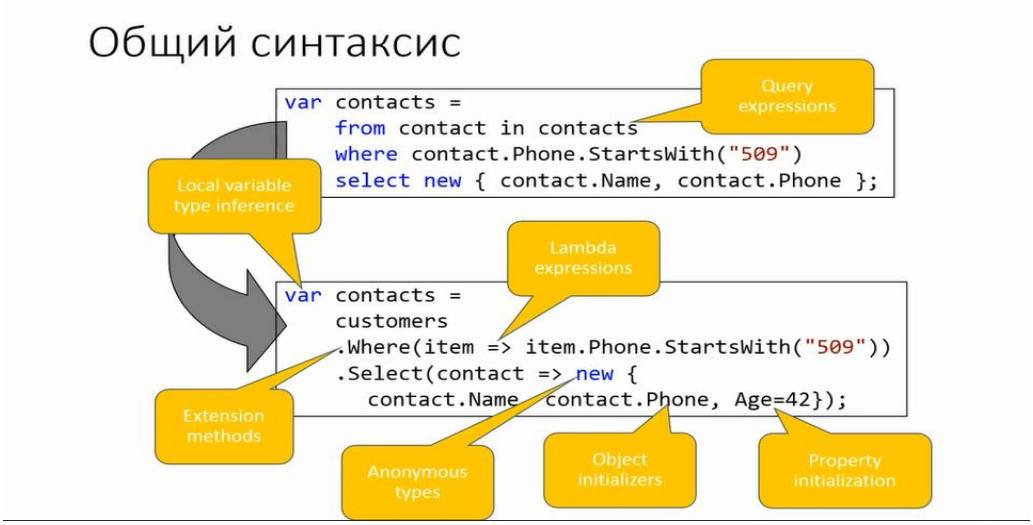
Пример 10

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace Modul20
{
    class Program
    {
        static void Main(string[] args)
        {
            List<Person> p = new List<Person>();
            p.Add(new Person() { FirstName = "Sergey", LastName = "Baydachnyy", Age =
35, Address = "Kirova 121"});
            p.Add(new Person() { FirstName = "Viktor", LastName = "Baydachnyy", Age =
34, Address = "Kirova 121" });
            p.Add(new Person() { FirstName = "Sergey", LastName = "Poplavskiy", Age =
30, Address = "Lenina 121" });
            p.Add(new Person() { FirstName = "Margarita", LastName = "Ostapchuk", Age =
15, Address = "Bandera 13" });

            /* Метод Where() позволяет перебрать элементы коллекции под какие-то
условие. В параметр item мы можем сохранить все отфильтрованные элементы коллекции, то
есть элементы, которые соответствуют условию - item.Age <= 30. Метод Select() берет в
качестве параметра каждый элемент, который соответствует нашему критерию в блоке
Where(...) слева от нашего оператора =>, в нашем примере элемент "it" */
            var u = p.Where(item => item.Age <= 30).Select(it =>  new { Name =
String.Format("{0} {1}", it.FirstName, it.LastName), Age = it.Age });
            Console.WriteLine(u.ToList().Count);

            Console.ReadLine();
        }
    }

    class Person
    {
        public string FirstName { get; set; }
        public string LastName { get; set; }
        public int Age { get; set; }
        public string Address { get; set; }
    }
}
```

Результат: 2

Написанному нами в *Примере 10* коду существует аналогия. Эта конструкция гораздо понятнее:

Пример 11



Сначала пишем ключевое слово **from** (*Пример 11*), после него пишется переменная, в которой будет сохранятся очередная запись, и за ним с помощью ключевого слово **in** мы должны описать где нам необходимо выбирать данные:

var list = from item in p …

item – наша переменная;
p – список текущих данных (коллекция), в котором и будет проходить выборка.

То есть, с помощью ключевых слов from и in мы определим откуда мы будем выбирать данные (**p**), и куда их будем ложить (**item**). Далее имея имя_переменной, которая будет хранить текущий элемент (item), мы можем писать ключевое слово **where** и писать наше условие. Пример:

Пример 12

var list = from item in p where item.Age <= 30 …

И далее мы можем использовать ключевое слово **select**, чтобы провести выборку. В нашем примере (*Пример 11*) это будет “анонимный тип”, который будет содержать свойства Name, Age.

Заключая, после ключевого слово from мы пишем переменную куда будем ложить очередную запись. После ключевого слово **in** записывается имя источника данных из которого мы будем брать очередную запись. Затем с помощью ключевого слово where мы задаем условие. А затем select позволяет нам указать, что мы хотим выбрать. Вместо “анонимного типа” в нашем примере, мы можем написать просто имя элемента item. Пример:

Пример 13

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
```

```
namespace Modul20
{
    class Program
    {
        static void Main(string[] args)
        {
            List<Person> p = new List<Person>();
            p.Add(new Person() { FirstName = "Sergey", LastName = "Baydachnyy", Age =
35, Address = "Kirova 121"});
            p.Add(new Person() { FirstName = "Viktor", LastName = "Baydachnyy", Age =
34, Address = "Kirova 121" });
            p.Add(new Person() { FirstName = "Sergey", LastName = "Poplavskiy", Age =
30, Address = "Lenina 121" });
            p.Add(new Person() { FirstName = "Margarita", LastName = "Ostapchuk", Age =
15, Address = "Bandera 13" });

            var u = from item in p
                    where item.Age <= 30
                /* Здесь вместо "анонимного типа" мы можем написать просто item, то
есть если мы хотим получить список элементов, которые удолетворяют условие item.Age
<=30 */
                    select new { Name = String.Format("{0} {1}", item.FirstName,
item.LastName), Age = item.Age };


            Console.WriteLine(u.ToList().Count);

            Console.ReadLine();
        }
    }

    class Person
    {
        public string FirstName { get; set; }
        public string LastName { get; set; }
        public int Age { get; set; }
        public string Address { get; set; }
    }
}
```

Результат: 2

ГЛАВА 16. АСИНХРОННОЕ ПРОГРАММИРОВАНИЕ

Потоки и процессы – представляют из себя последовательность инструкций, которые должны выполняться в определенном порядке. Инструкции в отдельных потоках или процессах, однако могут выполняться параллельно.

Процессы существуют в операционной системе и соответствуют тому, что пользователи видят как программы или приложения.

Поток существует внутри процесса. По этой причине потоки иногда называются “облегченные процессы”. Каждый процесс состоит из одного или более потоков.

Существование нескольких процессов позволяет компьютеру “одновременно” выполнять несколько задач. **Существование нескольких потоков позволяет процессу разделять работу для параллельного выполнения**. На многопроцессорном компьютере процессы или потоки могут работать на разных процессорах. Это позволяет выполнять реально параллельную работу.

ОС Windows для приложения выделяет некую сущность, которая называется процесс, которая содержит внутри себя ресурсы, необходимые для работы приложения. В то же время, приложения для того чтобы построить асинхронную работу, для того чтобы запустить несколько работ одновременно использует такое понятие как **потоки**.

Классы для создания потоков в .Net Framework:

1) Не самый оптимальный способ создания потоков внутри .Net Framework – это использование класса **Thread** (устаревший). Его особенность состоит в том, что каждый раз когда мы создаем экземпляр этого класса, фактически ОС требует выделить новый поток, то есть внутри ОС выделяется новый поток, что занимает какое-то количество времени, требует большого количество ресурсов, и сам по себе класс Thread не очень функциональный. Класс Thread находится в пространстве имен **System.Threading**. У класса Thread существуют несколько вариантов конструктора. Первый конструктор имеет параметр **(ParameterizedThreadStart start)**, этот конструктор мы используем в том случае, когда мы хотим запускать метод, который принимает какой-то параметр. Следующий тип конструктора имеет параметр **(ThreadStart start)**, и этот конструктор мы используем в том случае если мы хотим запускать метод, который не принимает параметр. Здесь нам нужно создать метод, который соответствует делегату ThreadStart, и запаковать его в этот делегат и передать его в качестве параметра конструктору класса Thread. Но мы можем это все сделать с помощью лямбда-выражения. Здесь наш делегат не принимает никаких параметров, поэтому пишем пустые круглые скобочки, после пишем оператор “=>”, и после пишем в фигурных скобочках написать любой код, который хотим запустить параллельно в потоке (в нашем примере (*Пример 1*) **Console.Writeline(“Hello from thread”);**). Пример:

Пример 1

```
using System;
using System.Threading;

namespace Modul21
{
    class Program
    {
        static void Main(string[] args)
        {
            Thread t = new Thread(() => { Console.WriteLine("Hello from thread!"); });
        }
    }
}
```

В *Примере 1* мы создали параллельный поток. Также в *Примере 1* мы создали объект класса Thread, но он еще ничего не выполняет, а для того чтобы запустить наш поток мы используем метод **Start()**:

имя_объекта_класса_Thread.Start();

И здесь если мы используем параметризированный делегат, то в качестве параметра методу Start() мы передаем object. А если у нас многопараметренный делегат, то мы должны упаковать в какой-то новый объект. Ну а в *Примере 2* у нас нет никаких параметров, то есть наш метод не принимает никаких параметров (мы не используем параметризированный делегат), поэтому мы метод Start() вызываем без параметров. Пример:

Пример 2

```
using System;
using System.Threading;

namespace Modul21
{
    class Program
    {
        static void Main(string[] args)
        {
            // Здесь мы создали обьект класса Thread
            Thread t = new Thread(() => { Console.WriteLine("Hello from thread!"); });

            /* Для того, чтобы запустить наш поток используем метод Start():
имя_обьекта_класса_Thread.Start(); */
            t.Start();
        }
    }
}
```

Результат: Hello from thread!

В нашем примере запускается параллельный поток, а наш метод **Main()** продолжает работать дальше. Для того чтобы затормозить работу потока мы используем метод **Sleep()** класса Thread указав в параметрах время его сна в миллисекундах. В нашем примере мы тормозим поток метода Main() на 1000 миллисекунд. Пример:

Пример 3

```
using System;
using System.Threading;

namespace Modul21
{
    class Program
    {
        static void Main(string[] args)
        {
            // Здесь мы создали обьект класса Thread
            Thread t = new Thread(() => { Console.WriteLine("Hello from thread!"); });

            /* Для того, чтобы запустить наш поток используем метод Start():
имя_обьекта_класса_Thread.Start(); */
```

```
            t.Start();

            /* Здесь мы попытаемся затормозить работу метода Main() с помощью метода
Sleep() класса Thread */
            /* Метод Sleep() позволяет усыпить поток(в данном случае поток в котором
работает метод Main()) на указанное в параметрах время */
            Thread.Sleep(1000);

            // С помощью вывода этого текста проверим задержку в потоке метод Main()
            Console.WriteLine("Hello from Main");
        }
    }
}
```

Результат: Hello from thread! (*через некоторое время*) Hello from Main

В *Примере 3* сначала выводиться “Hello from thread” из созданного нами потоке, также выводится с небольшим опозданием благодаря нашему методу Sleep() “Hello from Main” с потока метода Main(). На самом деле потоки выполняются одновременно.

А теперь если мы положим метод Sleep() внутри метода, которая запускается параллельно к потоку метода Main(). Пример:

Пример 4

```
using System;
using System.Threading;

namespace Modul21
{
    class Program
    {
        static void Main(string[] args)
        {
            // Здесь мы создали обьект класса Thread
            Thread t = new Thread(() => { Thread.Sleep(1000); Console.WriteLine("Hello
form thread!"); });

            /* Для того, чтобы запустить наш поток используем метод Start():
имя_обьекта_класса_Thread.Start(); */
            t.Start();

            // С помощью вывода этого текста проверим задержку в потоке метод Main()
            Console.WriteLine("Hello from Main");

            Console.ReadLine();
        }
    }
}
```

Результат: Hello from Main (*через некоторое время*) Hello from thread!

В *Примере 4* сначала выводится “Hello from Main”, а потом с некоторой задержкой “Hello from thread”. Проблема использование класса Thread в том, что класс Thread не позволяет получить возвращаемое значение, и еще если мы будем много использовать этот класса Thread(), то мы будем перегружать систему создавая много потоков. То есть, потоки пожирают много системных ресурсов ОС, потому что для каждого потока выделяется некий объем памяти. Все это не очень хорошо:

Thread class

- **Неправильный способ создания потоков внутри .Net приложений**
  - **Требует много ресурсов.**
  - **Требует реализацию ThreadStart делегата для упаковки метода.**
  - **Требует создание отдельного класса для передачи параметров.**

На сегодняшний день класс Thread практически уже не используется. Более того, в Windows Store и Windows Phone приложениях они заблокированы. Но у класса Thread есть альтернативы: использование **Thread pool**. “В чем состоит идея?” – Идея состоит в том, что на уровне .Net Framework для всех приложений создается некая одна структура, которая позволяет держать некий **pool-потоков**, и управлять этим pool-потоком. Здесь идея состоит в том, что когда мы пытаемся создать новый поток, то нам не создается новый объект на базе Thread, а нам выдергивается уже существующий объект из pool-а, и внутри этого объекта (внутри этого потока, который лежит в pool-е) запускается наш код. Во-первых, это быстрее, вот вторых, если мы делаем много потоков, то часть нашего кода не будет перегружать ОС, потому что **CLR (…)** не позволит создавать потоков больше чем может потянуть наша ОС:

Thread pool и ThreadPool class

- **Каждая версия CLR имеет свой pool-потоков.**
- **Pool-потоков является общим для всех приложений.**
- **Количество потоков в pool-е зависит от состояния системы.**
- **Имеются следующие ограничения:**
  - **Непонятно когда операция была завершена.**
  - **Нет возможности вернуть значение.**

CLR умеет управлять ресурсами компьютера. Поэтому если у нас много потоков (больше чем может потянуть наша ОС), то наш код (поток) может какое-то время ждать освобождения потоков, то есть находиться в очереди, как только из pool-потоков освободиться какой-то поток (то есть выполниться метод), он тут же будет предоставлен следующему в очереди блоку кода, которую необходимо выполнить в параллельном потоке. **Pool-потоков в системе является общим для всех приложений**.

При использований класса **ThreadPool** нам не надо создавать объект этого класса. Для того чтобы запустить код в параллельном потоке и выбрать его из pool-а, нам необходимо вызвать метод **QueueUserWorkItem()**, этот метод принимает в качестве параметра ссылку на какой-то делегат. Этот делегат принимает в качестве параметра метод (то есть указывает), который возвращает void и в качестве параметров принимает object. То есть если мы здесь хотим использовать лямбда-выражения, мы должны написать метод, который принимает некий object в качестве параметра и ничего не возвращает. Пример:

Пример 5

```
using System;
using System.Threading;

namespace Modul21
```

```
{
    class Program
    {
        static void Main(string[] args)
        {
            // В нашем примере item является object-ом и нашим параметром
            /* И в фигурных скобках мы пишем метод, который будет выполняться в
параллельном потоке */
            ThreadPool.QueueUserWorkItem((item) => { Console.WriteLine("Hello from
thread!"); });
            Thread.Sleep(1000);

            // С помощью вывода этого текста проверим задержку в потоке метод Main()
            Console.WriteLine("Hello from Main");

            Console.ReadLine();
        }
    }
}
```

Результат: Hello from thread! (*через некоторое время*) Hello from Main

"Как работает метод **QueueUserWorkItem()**?" – Он берет готовый блок кода, которую мы передаем ему в качестве параметра, и размещает его один из потоков, который находиться внутри pool-a, а если все потоки заняты, то код может какое-то время подождать пока поток не освободиться. Если мы хотим указать и второй параметр типа object помимо item (в наше примере) мы должны поставить запятую после фигурных скобочек, и написать какой-нибудь объект. А если мы не хотим использовать лямбда-выражения в этой конструкций, то можно и без них. Для этого мы создаем статический метод, который ничего не возвращает (void) и в качестве параметра принимает параметры типа object, и вместо лямбда-выражения в параметрах метода QueueUserWorkItem() напишем наш делегат (**делегат WaitCallback()**). И в качестве параметров наш делегат будет принимать ссылку на наш статический метод. Пример:

Пример 6

```
using System;
using System.Threading;

namespace Modul21
{
    class Program
    {
        static void Main(string[] args)
        {
            /* В качестве параметров делегата WaitCallback мы пишем метод, который
будет выполняться в параллельном потоке */
            ThreadPool.QueueUserWorkItem(new WaitCallback(MyMethod));
            Thread.Sleep(1000);

            // С помощью вывода этого текста проверим задержку в потоке метод Main()
            Console.WriteLine("Hello from Main");
            Console.ReadLine();
        }

        static void MyMethod(object state)
        {
            Console.WriteLine("Hello from thread!");
        }
    }
```

```
}

Результат: Hello from thread! (*через некоторое время*)  Hello from Main
```

С C# 2.0 появилась возможность не писать полностью new WaitCallback(MyMethod), а написать просто имя_метода, то есть MyMethod:

Пример 7

```
using System;
using System.Threading;

namespace Modul21
{
    class Program
    {
        static void Main(string[] args)
        {
            /* В качестве параметров мы пишем метод, который будет выполняться в
параллельном потоке */
            ThreadPool.QueueUserWorkItem(MyMethod);
            Thread.Sleep(1000);

            // С помощью вывода этого текста проверим задержку в потоке метод Main()
            Console.WriteLine("Hello from Main");
            Console.ReadLine();
        }

        static void MyMethod(object state)
        {
            Console.WriteLine("Hello from thread!");
        }
    }
}

Результат: Hello from thread! (*через некоторое время*)  Hello from Main
```

Еще один способ создания асинхронных вызовов по отношению каким-то методам – **асинхронный вызов методов с помощью делегатов**. Данный подход сейчас используется редко, но он является одним из основных шаблонов, который все еще используются в некоторых базовых классах, например, в классах по работе с сетью. Также этот подход используется при обращении к веб-службам (если мы генерируем прокси-класс), и во многих других вещах. Он является более гибким, потому что в отличие от предыдущего метода – QueueUserWorkItem(), он позволяет передавать любое количество параметров, и он позволяет возвращать значение. То есть по большему счету этот подход работает на основе того же pool-потоков, но при этом он более эффективный с точки зрения передачи параметров и возвращаемых значений. Например, пусть у нас будет метод, который принимает в качестве параметра переменную типа string и которая возвращает значение типа int. “Как нам теперь запустить в параллельном потоке?” – Для этого первое, мы создаем делегат, который соответствует прототипу нашего метода, которую мы хотим запустить асинхронно. В *Примере 8*:

```
delegate int MyDelegate(string s);
```

А после нам необходимо создать какую-то переменную, которая будет иметь тип этого делегата, куда и мы прикрутим наш метод. В *Примере 8*:

Static MyDelegate refMethod;

Переменная refMethod – переменная типа нашего делегата, которая может хранить ссылку на наш метод MyMethod(). Теперь наш метод мы можем привязать к refMethod. В *Примере 8*:

refMethod = MyMethod; - вот это конструкция тоже самое, что: **refMethod = new MyDelegate(MyMethod);**

Пример:

Пример 8

```
using System;
using System.Threading;

namespace Modul21
{
    class Program
    {
        // Здесь мы запускаем метод MyMethod() в параллельном потоке
        /* Для этого мы создаем делегат, который соответствует прототипу нашего метода,
которую мы хотим запустить асинхронно */
        delegate int MyDelegate(string s);

        // Переменная типа делегата, куда мы и прикрутим наш метод.
        // Переменная refMethod может хранить ссылку на наш метод
        static MyDelegate refMethod;

        static void Main(string[] args)
        {
            // Здесь мы привязываем метод MyMethod() к переменной refMethod
            /* Эту конструкцию мы можем написать так: refMethod = new
MyDelegate(MyMethod); */
            refMethod = MyMethod;

            Thread.Sleep(1000);
            // С помощью вывода этого текста проверим задержку в потоке метод Main()
            Console.WriteLine("Hello from Main");
            Console.ReadLine();
        }

        /* Метод, который возвращает значение типа int, и принимает в качестве
параметра переменную типа string */
        // И который мы будем запускать в паралельном потоке
        static int MyMethod(string state)
        {
            Console.WriteLine("Hello from thread!");
            return state.Length;
        }
    }
}
```

Мы уже говорили, что можем вызвать делегат так:

```
имя_переменной_типаделегата();
```

Или используя метод Invoke():

```
имя_переменной_типаделегата.Invoke();
```

Ну здесь еще один интересный метод **BeginInvoke()**. BeginInvoke() позволяет запустить методы, которые привязаны к делегату асинхронно, то есть в параллельных потоках. “Что он в качестве параметров принимает?” – Первым параметром он принимает в качестве параметра то (аргумент), что надо передать методом на которых указывает наш делегат в качестве аргумента. Вторым параметром ссылку на делегат, который будет ассоциироваться с методом, который вызывается при окончании работы метода(-ов), на которого указывает делегат. То есть, у нас появляется возможность реагировать на завершение работы методов нашего делегата. Здесь мы можем создать отдельный метод и упаковать его в **async**-делегат:

```
new AsyncCallback();
```

Или же можно написать лямбда-выражение. Делегат AsyncCallback ссылается на метод, который должен вызываться при завершении соответствующей асинхронной операции. Делегат AsyncCallback указывает на метолы, которые возвращают void, и которые в качестве параметра принимают некий **IAsyncResult**, то есть один параметр. Давайте мы напишем соответствующий лямбда-выражение этому делегату: 1) Сначала мы в круглых скобочках прописываем параметр (в нашем примере (*Пример 9*) result), который будет передаваться методу, который будет вызван во время окончание методов, на которых указывает наш делегат (в *Примере 9* после окончании метода MyMethod()), также он необходим для того чтобы вернуть значение, то есть для того чтобы получить статус потоков, и для того чтобы получить информацию о том, что были ли выполнены методы, за которыми должны выполняться лямбда-выражение. Далее в фигурных скобках прописываем ссылку на наш делегат (в *Примере 9* refMethod), и вызовем у него метод **EndInvoke()**. Метод EndInvoke() в качестве параметра принимает IAsyncResult (в *Примере 9* это – result), и этот метод возвращает значение типа данных, что и у нашего метода (соответственно как и у делегата), в *Примере 9* int. Третьим параметром метод BeginInvoke() принимает параметр, который позволяет передать ссылку на некий объект (этим объектом в нашем примере мог бы быть refMethod, но мы передаем просто null). Заключение:

1) Мы создаем делегат (в *Примере 9* MyDelegate), описывая его по соответствию прототипа нашего метода.

2) Создаем ссылку на делегат, который будет способна хранить данные о методах делегата, и которых мы хотим запускать в параллельных потоках. И запускаем его в параллельном потоке с помощью метода BeginInvoke(). BeginInvoke() позволяет первым параметром передать аргумент методу, на который указывает делегат, и который мы хотим запускать в параллельном потоке. Вторым параметром идет ссылка на делегат, который позволяет запустить метод, реагирующий на завершении нашего основного метода в

параллельном потоке, мы его описали как лямбда-выражение. Второй параметр (в нашем примере лямбда-выражение) запускается тоже в параллельном потоке. “Что это означает?” – Это означает, что если мы подобный подход будем использовать для того чтобы обновить графический интерфейс, то вот отсюда графический интерфейс обновлять нельзя, потому что графическим интерфейсом владеет интерфейсный (основной поток) поток нашего приложения и он не позволит нам модифицировать графический интерфейс из асинхронного (параллельного) потока. Здесь для того чтобы взаимодействовать основной поток с асинхронным нам нужно создать событие в асинхронном потоке. Пример для всего выше сказанной информации:

Пример 9

```
using System;
using System.Threading;

namespace Modul21
{
    class Program
    {
        // Здесь мы запускаем метод MyMethod() в параллельном потоке
        /* Для этого мы создаем делегат, который соответствует прототипу нашего метода,
которую мы хотим запустить асинхронно */
        delegate int MyDelegate(string s);

        /* Переменная типа делегата, куда мы и прикрутим наш метод. Переменная
refMethod может хранить ссылку на наш метод */
        static MyDelegate refMethod;

        static void Main(string[] args)
        {
            /* Здесь мы привязываем метод MyMethod() к переменной refMethod. Эту
конструкцию мы можем написать так: refMethod = new MyDelegate(MyMethod); */
            refMethod = MyMethod;

            refMethod.BeginInvoke("Hello", (result) => { int res =
refMethod.EndInvoke(result); Console.WriteLine(res); }, null);

            Thread.Sleep(1000);
            // С помощью вывода этого текста проверим задержку в потоке метод Main()
            Console.WriteLine("Hello from Main");
            Console.ReadLine();
        }

        /* Метод, который возвращает значение типа int, и принимает в качестве
параметра переменную типа string. И который мы будем запускать в паралельном потоке */
        static int MyMethod(string state)
        {
            Console.WriteLine("Hello from thread!");
            return state.Length;
        }
    }
}
```

Результат: Hello from thread; 5 Hello from Main

Как было уже сказано для того чтобы обновлять графические интерфейсы, нам необходимо отправить событие в интерфейсный (основной) поток. Для этого нужно получить ссылку на интерфейсный поток либо у всего нашего интерфейса, либо через конкретный элемент управления ссылку интерфейсный поток с помощью специального свойства, который называется **Dispatcher**. И как только мы получаем ссылку на

интерфейсный поток, мы можем туда бросить какое-то событие, то есть мы можем инициализировать (создать) событие в интерфейсном потоке через Dispatcher, внутри этого событие записать метод с помощью лямбда-выражений. Например, событие который будет обновлять интерфейсный поток. Если здесь наш метод небольшой то мы можем писать через лямбда-выражение наш метод, а если нам нужно написать много кода, и мы хотим отдельно вынести метод (который связан с интерфейсным потоком), то нам необходимо не только вызвать метод BeginInvoke(), но и нам необходимо задать делегат, и внутри этого делегата запаковать наш метод когда мы будем писать подобные вещи в Windows Forms. Windows 8. WPF приложениях, то мы должны об этом помнить:

Работа с интерфейсом

Если Вы решили обновить интерфейс не из интерфейсного потока, то:

- **Получите Dispatcher для интерфейсного потока.**
- **С помощью BeginInvoke() инициализируйте обновление.**

```
lblTime.Dispatcher.BeginInvoke(new Action (() => SetTime(currentTime)));
```

Последняя разработка в C#, работающая с потоками – использование класса Task. Класс Task облегчает работу с методами в параллельном потоке и он позволяет вернуть значения из параллельного потока и предоставляет несколько удобных методов при работе с параллельным потоком. Класс Task имеет конструктор. И в качестве параметра при создании объекта класса мы передаем ссылку на наш метод. Как только мы создаем экземпляр класса Task, мы можем использовать его методы. Один из его методов – это метод **Start()**. Этот метод позволяет запустить метод в параллельном потоке. Также методы в параллельном потоке можно запустить с помощью статических методов – **Task.Run()** и **Task.Factory.StartNew()**. Также есть набор интересных методов, которые позволяют дождаться в потоке завершение задачи, при этом это можно делать без блокирование интерфейсного потока. То есть фактически код прерывается, выполняется что-то в параллельном потоке, и затем код продолжает выполнение с того же места где мы и вызвали эти методы ожидания. Методы ожидания завершения задачи:

1) **Task.Wait()** блокирует работу нашего интерфейса, но ее также можно использовать паттерно, который бы не блокировал работу интерфейса. И в C# 4.0 этот паттерн реализован.

2) **Task.WaitAll()**.

3) **Task.WaitAny()**.

Управление задачами

- **Запуск:**
  - **Task.Start() – экземплярный метод;**
  - **Task.Factory.StartNew() – статический метод;**
  - **Task.Run() – статический метод.**

- **Ожидание завершения задачи:**
  - **Task.Wait();**
  - **Task.WaitAll();**
  - **Task.WaitAny().**

Делегаты

- **Асинхронный вызов делегатов:**
  - **BeginInvoke();**
  - **EndInvoke();**
  - **Обновление UI из интерфейсного потока.**
- **Шаблон используется в WebServices, WP 7 apps etc.**

Если мы хотим возвращаемое значение, то мы используем параметризированный класс **Task<TResult>**, то есть обобщенный класс, который в качестве параметров принимает тип, которую мы хотим возвращать из нашего метода. Пример:

Пример 10

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

namespace Modul21
{
    class Program
    {
        static void Main(string[] args)
        {
            /* Здесь мы создаем обьект параметризованного параметром int класса Task, так как наш метод(MyMethod()) возвращает значение типа int. И в качестве параметров класс Task принимает ссылку на функцию различных типов. В нашем примере наш метод MyMethod() получает в качестве параметра параметр типа string, и возвращает значение типа int. А такого конструктора у класса Task нету. Поэтому мы преобразуем параметр нашего метода с типа string на тип object. В параметрах обьекта класса Task мы прописываем наш метод, и вторым параметром пишем аргумент которую мы хотим передать методу MyMethod() при его вызове. Третьи параметр это CancellationToken - это специальный обьект, которую мы можем создать перед созданием обьекта класса Task, и этот обьект может в дальнейшем использоватся чтобы определить внутри асинхронного метода хотим ли мы ждать его окончание выполнение. То есть CancellationToken - обьект, который позволяет нам где-то внутри кода при нажатии к примеру пользователем на кнопку Cancel сказать "CancellationToken "закэнсели"(закрываться) пожалуйста асинхронный параллельный поток(асинхронное выполнение методов). Но при этом нам надо в нашем "методе в асинхронном потоке" явно написать код, который будет каждый раз опрашивать CancellationToken, и если CancellationToken говорит о том, что надо "кэнселится"(закрываться), то мы должны запустить код, закрывающий наш метод(выбросить исключение). CancellationToken не обязывает метод в параллельном потоке, потому что если он не поддерживает работу CancellationToken, то ничего и не будет. */

            Task<int> t = new Task<int>(MyMethod, "Hello");

            /* Здесь мы вызываем метод Start() у обьекта класса Task - t, который позволит запустить метод MyMethod() в параллельном потоке */
            t.Start();
```

```
  /* Здесь мы вызываем метод Wait() у обьекта класса Task - t, который дождется
окончания работы потока, и только после этого перейдет выдаче сообщение "Hello from
Main" */
            t.Wait();

            /* После того параллельный поток мы можем получить возвращаемое значение
метода в параллельном потоке. В нашем примере t.Result */
            Console.WriteLine(t.Result);

            Console.WriteLine("Hello from Main");
            Console.ReadLine();
        }

        /* Метод, который возвращает значение типа int, и принимает в качестве
параметра переменную типа string. И который мы будем запускать в паралельном потоке */
        static int MyMethod(object state)
        {
            Console.WriteLine("Hello from thread!");
            /* Здесь для того, чтобы получить доступ к методу класса string мы должны
преобразовать класс object в класс string с помощью ключевого слово as */
            return (state as string).Length;
        }
    }
}
```

Результат: Hello from thread! 5 Hello from Main

В .Net Framework существует библиотека, которая позволяет выполнять многие интересные вещи с помощью специального API. Там достаточно много методов, и часть из них используется для одновременного запуска нескольких задач (нескольких методов), для запуска цикла for в многопоточном режиме, для запуска цикла foreach в многопоточном режиме. Все эти вещи позволяют нам использовать многопоточное программирование при вычислениях, когда у нас есть несвязанные задачи, когда нам нужно параллельно выполнять сразу несколько задач одновременно:

Parallel API

- **Используйте Parallel.Invoke() для одновременного запуска нескольких задач:**

```
Parallel.Invoke(() => MethodForFirstTask(), () => MethodForSecondTask(), () =>
MethodForThirdTask());
```

- **Используйте Parallel.For для запуска цикла for в многопоточном режиме.**
- **Используйте Parallel.ForEach дл запуска цикла foreach в многопоточном режиме.**

При обработке исключений нам нужно понимать где это исключение перехватывать, то есть если мы хотим перехватить исключение из параллельного потока, то это возможно сделать только в том случае, если мы используем метод **Wait()**.

Обработка исключений

Вызывайте Task.Wait() для перехвата исключений. Перехватывайте AggregateException:

```
try
{
  task1.Wait();
}
catch (AggregateException)
{
  Foreach(var inner in ae.InnerExceptions)
  {
    // Deal with each exception in turn
  }
}
```

С помощью класса Task в C# 4.0 реализован подход, который еще больше упростил запуск параллельных методов. Этот подход базируется на двух ключевых словах: async и await, которые по большему счету позволяют выполнить метод внутри параллельного потока без большого количества сложных конструкций позволяют дождаться его результата:

Использование async и await

- **async модификатор в декларации метода**
- **await оператор применяется к async методам для ожидания результата без блокирования потока.**

```
private async void btnLongOperation_Click (object sender, RoutedEventArgs e)
{
  …
  Task<string> task1 = Task.Run<string>(() =>
  {
    …
  }
}
lblResult.Content = await task1;
```

Создание await методов

- **Используйте async**
- **Метод должен возвращать Task или Task<T>**

Пример:

Пример 11

```
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Linq;
using System.Net.Http;
using System.Text;
using System.Threading.Tasks;
using System.Windows.Forms;
namespace Modul21_Continue
{
    public partial class Form1 : Form
    {
        public Form1()
        {
            InitializeComponent();
        }

        private void button1_Click(object sender, EventArgs e)
        {
            tBlock.Text = DateTime.Now.ToString();
        }

        private void Form1_Load(object sender, EventArgs e) {}
        /* Мы должны обьявить метод, в котором мы собираемся вызывать что-то асинхронно с ключевым словом "async". То есть, это будет означать, что внутри этого метод может быть методы, которые запускаются в асиххронном потоке, и эти методы требуют ожидания окончания их работы. То есть, сам по себе этот метод будет спроектирован .Net Framework-ом и вызван таким образом, чтобы он мог дернуть что-то в асинхронном потоке, дождаться результатов и затем получить какие-то данные. Ну а для того, чтобы вызвать метод и дождаться результата мы должны использовать специальное ключевое слово "await". В нашем примере мы используем его перед нашим методом GetStringAsync(). То есть вот этот код - "await web.GetStringAsync(new Uri("http://censor.net", UriKind.Absolute));" позволяет нам дождаться результата метода GetStringAsync() и вернуть значение */

        private async void dButton_Click(object sender, EventArgs e)
        {
            dButton.Enabled = false;

            /* Класс HttpClient предоставляет базовый класс для отправки HTTP-запросов и получения HTTP-ответов от ресурса с заданным URI. Класс HttpClient позволяет выполнить загрузку контента */
            HttpClient web = new HttpClient();

            /* Метод GetStringAsync() позволяет вернуть наши данные с какого-то контента(к примеру, с сайта). Этот метод в качестве параметра принимает ссылку на URI. То есть он принимает ссылку на некий обьект типа URI. А ссылка на некий обьект типа URI в свою очередь принимает в качестве первого параметра принимает ссылку на какой-то контент(сайт), а вторым параметром прописываем типа адреса(UriKind.Absolute). То есть, метод GetStringAsync() будет загружать наши данные из контента(сайта) и возвращать эти данные, и тут мы присваиваем возвращаемое значение переменной типа string – s */
    string s = await web.GetStringAsync(new Uri("http://censor.net",
UriKind.Absolute));

            /* Но если мы будем делать все это синхронно, то он бы блокировал бы у нас интерфейсный поток. Поэтому в Windows Runtime для Windows 8 большинство методов(весь API)
```

```
обьявлено как асинхронные. "Что это означает?"-Это означает, что большинство методов(весь API) обьявлены с помощью ключевого слово "async", и все методы подразумевают возможность использование их в параллельных потоках. Это удобно, когда мы не хотим чтобы наш интерфейс тормозил и ждал окончания метода. Примером такого метода является выше используемый нами метод GetStringAsync(), этот метод с одной стороны возвращает обьект типа Task, с другой стороны обьявлен с помощью ключевого слово "async" */
            dButton.Enabled = true;
        }}}
```

За счет того что мы объявим методы как async и await, метод с пометкой await не блокирует интерфейсный поток, и с помощью ключевого слово await мы дожидаемся от методов ответа. А если бы мы не поставили ключевое слово await мы бы не могли вернуть возвращаемое значение метода (GetStringAsync()), и методпрото запускался бы в параллельном потоке и наш вызывающий метод (в нашем примере метод dButton_Click) не ждал бы его результата. Это удобно когда нам надо дождаться значение в асинхронных потоках, получить их и использовать их в своем потоке. Мы рассмотрели пример класса HttpClient, и использование уже готовых методов. А если нам нужно вызвать свой метод в асинхронном потоке используя ту же самую идеологию async, await. То есть если мы хотим написать метод, которую можно было бы вызывать в параллельном потоке, то первое что нам нужно сделать – надо написать сам метод. Этот метод должен возвращать значение типа Task, тут можно писать Task без параметра Task или с параметром Task<T>. К примеру: Task<int>. И мы должны написать ключевое слово async перед Task<T>. Пример:

Пример 12

```
async Task<int> MyMethod(string s)
{
  return s.Length;
}
```

И теперь этот метод мы можем вызывать из метода, который объявлен с помощью ключевого слово async (как в нашем предыдущем примере с методом GetStringAsync()). И чтобы вызвать наш метод нужно просто написать его имя и перед его именем написать ключевое слово await. А если бы мы вызвали его (метод) просто по имени, то он просто выполнялся бы в параллельном потоке и не вернет нам значение. Когда мы вызываем наш метод по имени и с ключевым словом await, то мы ждем его результата (окончание, возвращаемое значение). Вот так мы и запускаем наш метод в параллельных потоках. Пример:

Пример 13

```
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Linq;
using System.Net.Http;
using System.Text;
using System.Threading.Tasks;
using System.Windows.Forms;

namespace Modul21_Continue
{
    public partial class Form1 : Form
    {
```

```
        public Form1()
        {
            InitializeComponent();
        }

        private void button1_Click(object sender, EventArgs e)
        {
            tBlock.Text = DateTime.Now.ToString();
        }

        private void Form1_Load(object sender, EventArgs e)
        {

        }

        private async void dButton_Click(object sender, EventArgs e)
        {
            dButton.Enabled = false;
            int res = await MyMethod("Hello");

            dButton.Enabled = true;
        }

        async Task<int> MyMethod(string s)
        {
            return s.Length;
        }
    }
}
```

Все что нам нужно было это описать метод запаковав возвращаемое значение в объект типа Task (в нашем примере Task<int>), а если метод ничего не возвращает, то нам бы было достаточно написать просто Task. Вот эта строчка – **… await MyMethod(“Hello”);** позволяет нам ожидать результат (возвращаемое значение) метода в параллельном потоке и вернуть значение. Для того чтобы можно было это делать метод должен иметь ключевое слово async.

async и await позволяют нам упростить механизм создания и запуска методов, которые выполняются в параллельном потоке. Для этого мы можем спокойно создавать любые методы, с любыми возвращаемыми значениями, с любыми параметрами. Но мы должны паковать возвращаемое значение во внутрь класса Task (Task и Task<T>). Для вызова таких методов мы используем ключевые слова async и await. async необходим для того чтобы дать понять компилятору что этот метод внутри себя может содержать вызов асинхронных методов, и этот метод будет использоваться для того, чтобы дождаться ответов вот этих асинхронных методов. await используется для того чтобы дать понять “вот есть метод, и он работает, давайте дождемся его результатов, и когда он вернет значение мы можем продолжать дальше”.

Делегаты и Task

- **Используйте TaskFactory.FromAsync для работы с методами по принципу делегатов.**

  HttpWebRequest request = (HttpWebRequest) WebRequest.Create(url);

HttpWebResponse response = await Task<WebResponse>.Factory.FromAsync (request.BeginGetResponse, request.EndGetResponse) as HttpWebResponse;

Класс **TaskFactory** имеет метод **FromAsync**, который позволяет работать с тем подходом, которую мы рассматривали, то есть с подходом на примере делегатов. Он позволяет работать с этим подходом просто, то есть он позволяет не писать множество методов, а позволяет запихнуть в вызов одного метода, и получить сразу нужное значение дождавшись окончания работы самого последнего метода, который возвращает результат. Он может быть удобен при работе с веб-службами, или при работе со старым API, который уже был написан разработчиком, который не хочет переписывать через Task.

Использование блокировок

```
private object lockingObject = new object();
lock (lockingObject)
{
  //Only one thread can enter this block at any one time
}
```

Когда у нас к одному и тому же объекту будут пытаться получить доступ сразу несколько потоков, и если там изменение сложные, то нам нужно предварительно эти объекты заблокировать. “Как это делается?” – Обычно выделяется те куски кода, которые не могут быть использованы одновременно несколькими потоками, создается некий приватный объект (**private object lockingObject = new object();**), и используется специальное ключевое слово – **lock**. lock – самый примитивный блокиратор в C#. В круглых скобках рядом с lock мы пишем объект, который будет блокироваться. А после в фигурных скобках пишется тот код, который должен быть использован только одним потоком одновременно, то есть как только другой поток дойдет до этого кода, он в момент попадания на lock начинает выполнять ожидания окончание работы кода, потому что он не может его выполнить до тех пор, пока объект (в нашем примере lockingObject) не будет освобожден. Как только объект будет свободным следующий поток может приступать к его выполнению. В .Net Framework существует масса других классов, которые позволяют управлять синхронизацией потоков.

Классы для синхронизации в *TaskParallelLibrary*

- **Use the *ManualResetEventSlim* class to limit resource access to one thread at a time.**
- **Use the *SemaphoreSlim* class to limit resource access to a fixed number of threads.**
- **Use the *CountdownEvent* class to block a thread until a fixed number of tasks signal completion.**
- **Use the *ReaderWriterLockSlim* class to allow multiple threads to read a resource or a single thread to write to a resource at any one time.**
- **Use the *Barrier* class to block multiple threads until they all satisfy a condition.**

Существуют также безопасные коллекции в .Net Framework, которые адаптированы для работы с параллельными потоками:

Безопасные коллекции

- **ConcurrentBag<T>**
- **ConcurrentDictionary<TKey, TValue>**
- **ConcurrentQueue<T>**
- **ConcurrentStack<T>**
- **IProducerConsumerCollection<T>**
- **BlockingCollection<T>**