# Csc.exe compiler

Используем компилятор. Для этого откроем командную строку. С помощью команды

cd C:\csharp

Перейдем к папке, в которой хранится файл Program.cs.

Далее передадим этот файл компилятору с помощью команды (для C# 6):

"C:\Program Files (x86)\Microsoft Visual Studio\2017\Community\MSBuild\15.0\Bin\Roslyn" Program.cs

Поскольку название папки Program Files состоит из нескольких слов, то весь путь к ней берется в кавычки. А после пробела указывается название файла программы.

Если на компьютере установлена VS 2013 или более старая версия (то есть C# 5/4/3.5), то соответственно используется другая команда:

C:\Windows\Microsoft.NET\Framework\v4.0.30319\csc.exe Program.cs

# Поразрядные операции

Особый класс операций представляют поразрядные операции. Они выполняются над отдельными разрядами числа. В этом плане числа рассматриваются в двоичном представлении, например, 2 в двоичном представлении 10 и имеет два разряда, число 7 - 111 и имеет три разряда.

### Логические операции

* **&**(логическое умножение)

Умножение производится поразрядно, и если у обоих операндов значения разрядов равно 1, то операция возвращает 1, иначе возвращается число 0. Например:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | int x1 = 2; //010   int y1 = 5;//101   Console.WriteLine(x1&y1); // выведет 0    int x2 = 4; //100  int y2 = 5; //101  Console.WriteLine(x2 & y2); // выведет 4 |

В первом случае у нас два числа 2 и 5. 2 в двоичном виде представляет число 010, а 5 - 101. Поразрядно умножим числа (0\*1, 1\*0, 0\*1) и в итоге получим 000.

Во втором случае у нас вместо двойки число 4, у которого в первом разряде 1, так же как и у числа 5, поэтому в итоге получим (1\*1, 0\*0, 0 \*1) = 100, то есть число 4 в десятичном формате.

* **|** (логическое сложение)

Похоже на логическое умножение, операция также производится по двоичным разрядам, но теперь возвращается единица, если хотя бы у одного числа в данном разряде имеется единица. Например:

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | int x1 = 2; //010  int y1 = 5;//101  Console.WriteLine(x1|y1); // выведет 7 - 111  int x2 = 4; //100  int y2 = 5;//101  Console.WriteLine(x2 | y2); // выведет 5 - 101 |

* **^** (логическое исключающее ИЛИ)

Также эту операцию называют XOR, нередко ее применяют для простого шифрования:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | int x = 45; // Значение, которое надо зашифровать - в двоичной форме 101101  int key = 102; //Пусть это будет ключ - в двоичной форме 1100110  int encrypt = x ^ key; //Результатом будет число 1001011 или 75  Console.WriteLine("Зашифрованное число: " +encrypt);    int decrypt = encrypt ^ key; // Результатом будет исходное число 45  Console.WriteLine("Расшифрованное число: " + decrypt); |

Здесь опять же производятся поразрядные операции. Если у нас значения текущего разряда у обоих чисел разные, то возвращается 1, иначе возвращается 0. Таким образом, мы получаем из 9^5 в качестве результата число 12. И чтобы расшифровать число, мы применяем ту же операцию к результату.

* **~** (логическое отрицание или инверсия)

Еще одна поразрядная операция, которая инвертирует все разряды: если значение разряда равно 1, то оно становится равным нулю, и наоборот.

|  |  |
| --- | --- |
| 1  2 | int x = 9;  Console.WriteLine(~x); |

### Операции сдвига

Операции сдвига также производятся над разрядами чисел. Сдвиг может происходить вправо и влево.

* x<<y - сдвигает число x влево на y разрядов. Например, 4<<1 сдвигает число 4 (которое в двоичном представлении 100) на один разряд влево, то есть в итоге получается 1000 или число 8 в десятичном представлении.
* x>>y - сдвигает число x вправо на y разрядов. Например, 16>>1 сдвигает число 16 (которое в двоичном представлении 10000) на один разряд вправо, то есть в итоге получается 1000 или число 8 в десятичном представлении.

Таким образом, если исходное число, которое надо сдвинуть в ту или другую строну, делится на два, то фактически получается умножение или деление на два. Поэтому подобную операцию можно использовать вместо непосредственного умножения или деления на два.

# Потеря данных и ключевое слово checked

Рассмотрим другую ситуацию, что будет, например, в следующем случае:

|  |  |
| --- | --- |
| 1  2  3 | int a = 33;  int b = 600;  byte c = (byte)(a+b); |

Результатом будет число 121, так число 633 не попадает в допустимый диапазон для типа byte, и старшие биты будут усекаться. В итоге получится число 121. Поэтому при преобразованиях надо это учитывать. И мы в данном случае можем либо взять такие числа a и b, которые в сумме дадут число не больше 255, либо мы можем выбрать вместо byte другой тип данных, например, int.

Однако ситуации разные могут быть. Мы можем точно не знать, какие значения будут иметь числа a и b. И чтобы избежать подобных ситуаций, в c# имеется ключевое слово checked:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11 | try  {      int a = 33;      int b = 600;      byte c = checked((byte)(a + b));      Console.WriteLine(c);  }  catch (OverflowException ex)  {      Console.WriteLine(ex.Message);  } |

При использовании ключевого слова checked приложение выбрасывает исключение о переполнении. Поэтому для его обработки в данном случае используется конструкция try...catch. Подробнее данную конструкцию и обработку исключений мы рассмотрим позже, а пока надо знать, что в блок try мы включаем действия, в которых может потенциально возникнуть ошибка, а в блоке catch обрабатываем ошибку.

# исключающее ИЛИ

Операция исключающего ИЛИ. Возвращает true, если либо первый, либо второй операнд (но не одновременно) равны true, иначе возвращает false

|  |  |
| --- | --- |
| 1  2 | bool x5 = (5 > 6) ^ (4 < 6); // 5 > 6 - false, 4 < 6 - true, поэтому возвращается true  bool x6 = (50 > 6) ^ (4 / 2 < 3); // 50 > 6 - true, 4/2 < 3 - true, поэтому возвращается false |

# Ref Out Именованые И Необезательные параметры

### Необязательные параметры

C# позволяет использовать необязательные параметры. Для таких параметров нам необходимо объявить значение по умолчанию. Также следует учитывать, что после необязательных параметров все последующие параметры также должны быть необязательными:

|  |  |
| --- | --- |
| 1  2  3  4 | static int OptionalParam(int x, int y, int z=5, int s=4)  {      return x + y + z + s;  } |

Так как последние два параметра объявлены как необязательные, то мы можем один из них или оба опустить:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | static void Main(string[] args)  {      OptionalParam(2, 3);        OptionalParam(2,3,10);        Console.ReadLine();  } |

### Именованные параметры

В предыдущих примерах при вызове методов значения для параметров передавались в порядке объявления этих параметров в методе. Но мы можем нарушить подобный порядок, используя именованные параметры:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13 | static int OptionalParam(int x, int y, int z=5, int s=4)  {      return x + y + z + s;  }  static void Main(string[] args)  {      OptionalParam(x:2, y:3);        //Необязательный параметр z использует значение по умолчанию      OptionalParam(y:2,x:3,s:10);        Console.ReadLine();  } |

# Params

Во всех предыдущих примерах мы использовали постоянное число параметров. Но, используя ключевое слово **params**, мы можем передавать неопределенное количество параметров:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20 | static void Addition(params int[] integers)  {      int result = 0;      for (int i = 0; i < integers.Length; i++)      {          result += integers[i];      }      Console.WriteLine(result);  }    static void Main(string[] args)  {      Addition(1, 2, 3, 4, 5);        int[] array = new int[] { 1, 2, 3, 4 };      Addition(array);        Addition();      Console.ReadLine();  } |

Причем, как видно из примера, на место параметра с модификатором params мы можем передать как отдельные значения, так и массив значений, либо вообще не передавать параметры.

Однако данный способ имеет ограничения: после параметра с модификатором params мы не можем указывать другие параметры. То есть следующее определение метода недопустимо:

|  |  |
| --- | --- |
| 1  2 | static void Addition(params int[] integers, int x, string mes)  {} |

# Сокрытие переменных

При работе с переменными надо учитывать, что локальные переменные, определенные в методе или в блоке кода, скрывают переменные уровня класса, если их имена совпадают:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10 | class Program  {      static int a = 9; // переменная уровня класса        static void Main(string[] args)      {          int a = 5; // скрывает переменную a, которая объявлена на уровне класса          Console.WriteLine(a); // 5      }  } |

При объявлении переменных также надо учитывать, что в одном контексте нельзя определить несколько переменных с одним и тем же именем.

# Кортежи

Кортежи предоставляют удобный способ для работы с набором значений, который был добавлен в версии C# 7.0.

Кортеж представляет набор значений, заключенных в круглые скобки:

|  |  |
| --- | --- |
| 1 | var tuple = (5, 10); |

В данном случае определен кортеж tuple, который имеет два значения: 5 и 10. В дальнейшем мы можем обращаться к каждому из этих значений через поля с названиями Item[порядковый\_номер\_поля\_в\_кортеже]. Например:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | static void Main(string[] args)  {      var tuple = (5, 10);      Console.WriteLine(tuple.Item1); // 5      Console.WriteLine(tuple.Item2); // 10      tuple.Item1 += 26;      Console.WriteLine(tuple.Item1); // 31      Console.Read();  } |

В данном случае тип определяется неявно. Но мы ткже можем явным образом указать для переменной кортежа тип:

|  |  |
| --- | --- |
| 1 | (int, int) tuple = (5, 10); |

Так как кортеж содержит два числа, то в определении типа нам надо указать два числовых типа. Или другой пример определения кортежа:

|  |  |
| --- | --- |
| 1 | (string, int, double) person = ("Tom", 25, 81.23); |

Первый элемент кортежа в данном случае представляет строку, второй элемент - тип int, а третий - тип double.

Мы также можем дать названия полям кортежа:

|  |  |
| --- | --- |
| 1  2  3 | var tuple = (count:5, sum:10);  Console.WriteLine(tuple.count); // 5  Console.WriteLine(tuple.sum); // 10 |

Теперь чтобы обратиться к полям кортежа используются их имена, а не названия Item1 и Item2.

Мы даже можем не использовать переменную для определения всего кортежа, а использовать отдельные переменные для его полей:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | static void Main(string[] args)  {      var (name, age) = ("Tom", 23);      Console.WriteLine(name);    // Tom      Console.WriteLine(age);     // 23      Console.Read();  } |

Например, одной из распространенных ситуаций является возвращение из функции двух и более значений, в то время как функция можно возвращать только одно значение. И кортежи представляют оптимальный способ для решения этой задачи:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13 | static void Main(string[] args)  {      var tuple = GetValues();      Console.WriteLine(tuple.Item1); // 1      Console.WriteLine(tuple.Item2); // 3        Console.Read();  }  private static (int, int) GetValues()  {      var result = (1, 3);      return result;  } |

Здесь определен метод GetValues(), который возвращает кортеж. Кортеж определяется как набор значений, помещенных в круглые скобки. И в данном случае мы возвращаем кортеж из двух элементов типа int, то есть два числа.

Другой пример:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18 | static void Main(string[] args)  {      var tuple = GetNamedValues(new int[]{ 1,2,3,4,5,6,7});      Console.WriteLine(tuple.count);      Console.WriteLine(tuple.sum);        Console.Read();  }  private static (int sum, int count) GetNamedValues(int[] numbers)  {      var result = (sum:0, count: 0);      for (int i=0; i < numbers.Length; i++)      {          result.sum += numbers[i];          result.count++;      }      return result;  } |

И также кортеж может передаваться в качестве параметра в метод:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13 | static void Main(string[] args)  {      var (name, age) = GetTuple(("Tom", 23), 12);      Console.WriteLine(name);    // Tom      Console.WriteLine(age);     // 35      Console.Read();  }    private static (string name, int age) GetTuple((string n, int a) tuple, int x)  {      var result = (name: tuple.n, age: tuple.a + x);      return result;  } |

# Оператор throw

Чтобы сообщить о выполнении исключительных ситуаций в программе, можно использовать оператор **throw**. То есть с помощью этого оператора мы сами можем создать исключение и вызвать его в процессе выполнения. Например, в нашей программе происходит ввод строки, и мы хотим, чтобы, если длина строки будет больше 6 символов, возникало исключение:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16 | static void Main(string[] args)  {      try      {          string message = Console.ReadLine();          if (message.Length > 6)          {              throw new Exception("Длина строки больше 6 символов");          }      }      catch (Exception e)      {          Console.WriteLine("Ошибка: " + e.Message);      }      Console.ReadLine();  } |

# Фильтры исключений

В C# 6.0 (Visual Studio 2015) была добавлена такая функциональность, как фильтры исключений. Они позволяют обрабатывать исключения в зависимости от определенных условий:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15 | int x = 1;  int y = 0;    try  {      int result = x / y;  }  catch(Exception ex) when (y==0)  {      Console.WriteLine("y не должен быть равен 0");  }  catch(Exception ex)  {      Console.WriteLine(ex.Message);  } |

# Локальная переменная-ссылка

Для определения локальной переменной-ссылки (ref local) перед ее типом ставится ключевое слово **ref**:

|  |  |
| --- | --- |
| 1  2 | int x = 5;  ref int xRef = ref x; |

Здесь переменная xRef указывает не просто на значение переменной x, а на область в памяти, где располагается эта переменная. Для этого перед x также указывается ref.

При этом мы не можем просто определить переменную-ссылку, нам обязательно надо присвоить ей некоторое значение. Так, следующий код вызовет ошибку:

### Ссылка как результат функции

Для возвращения из функции ссылки в сигнатуре функции перед возвращаемым типом, а также после оператора **return** следует указать ключевое слово **ref**:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20 | static void Main(string[] args)  {      int[] numbers = { 1, 2, 3, 4, 5, 6, 7 };      ref int numberRef = ref Find(4, numbers); // ищем число 4 в массиве      numberRef = 9; // заменяем 4 на 9      Console.WriteLine(numbers[3]); // 9        Console.Read();  }  static ref int Find(int number, int[] numbers)  {      for (int i = 0; i < numbers.Length; i++)      {          if (numbers[i] == number)          {              return ref numbers[i]; // возвращаем ссылку на адрес, а не само значение          }      }      throw new IndexOutOfRangeException("число не найдено");  } |

# Модификаторы доступа

В C# применяются следующие модификаторы доступа:

* **public**: публичный, общедоступный класс или член класса. Такой член класса доступен из любого места в коде, а также из других программ и сборок.
* **private**: закрытый класс или член класса. Представляет полную противоположность модификатору public. Такой закрытый класс или член класса доступен только из кода в том же классе или контексте.
* **protected**: такой член класса доступен из любого места в текущем классе или в производных классах.
* **internal**: класс и члены класса с подобным модификатором доступны из любого места кода в той же сборке, однако он недоступен для других программ и сборок (как в случае с модификатором public).
* **protected internal**: совмещает функционал двух модификаторов. Классы и члены класса с таким модификатором доступны из текущей сборки и из производных классов.

# Константы

Особенностью констант является то, что их значение можно установить только один раз. Например, если у нас в программе есть некоторые переменные, которые не должны изменять значение (к примеру, число PI, число e и т.д.), мы можем объявить их константами. Для этого используется ключевое слово **const**:

|  |  |
| --- | --- |
| 1  2 | const double PI = 3.14;  const double E = 2.71; |

# Поля для чтения

Поля для чтения можно инициализировать при их объявлении либо на уровне класса, либо инициилизировать и изменять в конструкторе. Инициализировать или изменять их значение в других местах нельзя, можно только считывать их значение.

Поле для чтения объявляется с ключевым словом **readonly**:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12 | class MathLib  {      public readonly double K = 23;  // можно так инициализировать        public MathLib(double \_k)      {          K = \_k; // поле для чтения может быть инициализировано или изменено в конструкторе после компиляции      }      public void ChangeField()      {          // так нельзя          //K = 34;      } |

# Свойства

class Person

{

    private int age;

    public int Age

    {

        set

        {

            if (value < 18)

            {

                Console.WriteLine("Возраст должен быть больше 17");

            }

            else

            {

                age = value;

            }

        }

        get { return age; }

    }

}

### Инициализация автосвойств

Начиная с версии C# 6.0 была добавлена такая функциональность, как инициализация автосвойств:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17 | class Person  {      public string Name { get; set; } = "Tom";      public int Age { get; set; } = 23;  }    class Program  {      static void Main(string[] args)      {          Person person = new Person();          Console.WriteLine(person.Name); // Tom          Console.WriteLine(person.Age);  // 23            Console.Read();      }  } |

И если мы не укажем для объекта Person значения свойств Name и Age, то будут действовать значения по умолчанию.

Еще одно изменение коснулось определения автосвойств. Например, если в C# 5.0 мы захотели сделать автосвойство доступным для установки только из класса, то надо было указать **private set**:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | class Person  {      public string Name { get; private set; }      public Person(string n)      {          Name = n;      }  } |

Кроме как из класса Person это свойство невозможно установить. Начиная с версии C# 6.0 нам необязательно писать private set, мы можем оставить только выражение get:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | class Person  {      public string Name { get;}      public Person(string n)      {          Name = n;      }  } |

# Статические члены класса

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26 | class Account  {      public Account(decimal sum, decimal rate)      {          if (sum < MinSum) throw new Exception("Недопустимая сумма!");          Sum = sum; Rate = rate;      }      private static decimal minSum = 100; // минимальная допустимая сумма для всех счетов      public static decimal MinSum      {          get { return minSum; }          set { if(value>0) minSum = value; }      }        public decimal Sum { get; private set; }    // сумма на счете      public decimal Rate { get; private set; }   // процентная ставка        // подсчет суммы на счете через определенный период по определенной ставке      public static decimal GetSum(decimal sum, decimal rate, int period)      {          decimal result = sum;          for (int i = 1; i <= period; i++)              result = result + result \* rate / 100;          return result;      }  } |

### Статический конструктор

Кроме обычных конструкторов у класса также могут быть статические конструкторы. Статические конструкторы выполняются при самом первом создании объекта данного класса или первом обращении к его статическим членам (если таковые имеются):

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17 | class User  {      static User()      {          Console.WriteLine("Создан первый пользователь");      }  }  class Program  {      static void Main(string[] args)      {          User user1 = new User(); // здесь сработает статический конструктор          User user2 = new User();            Console.Read();      }  } |

### Статические классы

Статические классы объявляются с модификатором static и могут содержать только статические поля, свойства и методы. Например, если бы класс Account имел бы только статические переменные, свойства и методы, то его можно было бы объявить как статический:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18 | static class Account  {      private static decimal minSum = 100; // минимальная допустимая сумма для всех счетов      public static decimal MinSum      {          get { return minSum; }          set { if(value>0) minSum = value; }      }        // подсчет суммы на счете через определенный период по определенной ставке      public static decimal GetSum(decimal sum, decimal rate, int period)      {          decimal result = sum;          for (int i = 1; i <= period; i++)              result = result + result \* rate / 100;          return result;      }  } |

## Перегрузка операторов

class Counter

{

    public int Value { get; set; }

    public static Counter operator +(Counter c1, Counter c2)

    {

        return new Counter { Value = c1.Value + c2.Value };

    }

    public static bool operator >(Counter c1, Counter c2)

    {

        if (c1.Value > c2.Value)

            return true;

        else

            return false;

    }

    public static bool operator <(Counter c1, Counter c2)

    {

        if (c1.Value < c2.Value)

            return true;

        else

            return false;

    }

}

## Перегрузка операций преобразования типов

После модификаторов **public static** идет ключевое слово **explicit** (если преобразование явное, то есть нужна операция приведения типов) или **implicit** (если преобразование неявное). Затем идет ключевое слово **operator** и далее возвращаемый тип, в который надо преобразовать объект. В скобках в качестве параметра передается объект, который надо преобразовать.

Например, пусть у нас есть следующий класс Counter, который представляет счетчик-секундомер и который хранит количество секунд в свойстве Seconds:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13 | class Counter  {      public int Seconds { get; set; }        public static implicit operator Counter(int x)      {          return new Counter { Seconds = x };      }      public static explicit operator int(Counter counter)      {          return counter.Seconds;      }  } |

Первый оператор преобразует число - объект типа int к типу Counter. Его логика проста - создается новый объект Counter, у которого устанавливается свойство Seconds.

Второй оператор преобразует объект Counter к типу int, то есть получает из Counter число.

Примение операторов преобразования в программе:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10 | static void Main(string[] args)  {      Counter counter1 = new Counter { Seconds = 23 };        int x = (int)counter1;      Console.WriteLine(x);   // 23        Counter counter2 = x;      Console.WriteLine(counter2.Seconds);  // 23  } |

## Абстрактные классы

Абстрактные методы также, как и виртуальные, являются частью полиморфного интерфейса. Но если в случае с виртуальными методами мы говорим, что класс-наследник наследует реализацию, то в случае с абстрактными методами наследуется интерфейс, представленный этими абстрактными методами.

Другим хрестоматийным примером является система геометрических фигур. В реальности не существует геометрической фигуры как таковой. Есть круг, прямоугольник, квадрат, но просто фигуры нет. Однако же и круг, и прямоугольник имеют что-то общее и являются фигурами:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30 | // абстрактный класс фигуры  abstract class Figure  {      // абстрактный метод для получения периметра      public abstract float Perimeter();      // абстрактный метод для получения площади      public abstract float Area();  }  // производный класс прямоугольника  class Rectangle : Figure  {      public float Width { get; set; }      public float Height { get; set; }        public Rectangle(float width, float height)      {          this.Width = width;          this.Height = height;      }      // переопределение получения периметра      public override float Perimeter()      {          return Width \* 2 + Height \* 2;      }      // переопрелеление получения площади      public override float Area()      {          return Width \* Height;      }  } |

## Преобразование типов

Рассмотрим некоторые примеры преобразований:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13 | // Объект Employee также представляет тип object  object emp = new Employee("Bill", "Gates", "Microsoft");  // объект Client также представляет тип Person  Person cl = new Client("Tom", "Johnes", "SberBank", 200);    //у класса Object нет метода Display, поэтому приводим к классу Employee  ((Employee)emp).Display();    // у класса Person есть метод Display  cl.Display();    // у класса Person нет свойства Bank, поэтому приводим к классу Client  string bank = ((Client)cl).Bank; |

## Обобщения

|  |
| --- |
| class Account<T>  {      public T Id { get; set; }      public int Sum { get; set; }  } |

Угловые скобки в описании class Account<T> указывают, что класс является обобщенным, а тип T, заключенный в угловые скобки, будет использоваться этим классом. Необязательно использовать именно букву T, это может быть и любая другая буква или набор символов. Причем сейчас нам неизвестно, что это будет за тип, это может быть любой тип. Поэтому параметр **T** в угловых скобках в еще называется **универсальным параметром**, так как вместо него можно подставить любой тип.

Например, вместо параметра T можно использовать объект int, то есть число, представляющее номер счета. Это также может быть объект string, либо или любой другой класс или структура:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | Account<int> account1 = new Account<int> { Sum = 5000 };  Account<string> account2 = new Account<string> { Sum = 4000 };  account1.Id = 2;        // упаковка не нужна  account2.Id = "4356";  int id1 = account1.Id;  // распаковка не нужна  string id2 = account2.Id;  Console.WriteLine(id1);  Console.WriteLine(id2); |

### Значения по умолчанию

Иногда возникает необходимость присвоить переменным универсальных параметров некоторое начальное значение, в том числе и null. Но напрямую мы его присвоить не можем:

|  |  |
| --- | --- |
| 1 | T id = null; |

В этом случае нам надо использовать оператор **default(T)**. Он присваивает ссылочным типам в качестве значения null, а типам значений - значение 0:

|  |  |
| --- | --- |
| 1  2  3  4 | class Account<T>  {      T id = default(T);  } |

### Статические поля обобщенных классов

При типизации обобщенного класса определенным типом будет создаваться свой набор статических членов. Например, в классе Account определено следующее статическое поле:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | class Account<T>  {      public static T session;        public T Id { get; set; }      public int Sum { get; set; }  } |

Теперь типизируем класс двумя типами int и string:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | Account<int> account1 = new Account<int> { Sum = 5000 };  Account<int>.session = 5436;    Account<string> account2 = new Account<string> { Sum = 4000 };  Account<string>.session = "45245";    Console.WriteLine(Account<int>.session);      // 5436  Console.WriteLine(Account<string>.session);   // 45245 |

### Обобщенные методы

Кроме обобщенных классов можно также создавать обобщенные методы, которые точно также будут использовать универсальные параметры. Например:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23 | class Program  {      private static void Main(string[] args)      {          int x = 7;          int y = 25;          Swap<int>(ref x, ref y);          Console.WriteLine($"x={x}    y={y}");   // x=25   y=7            string s1 = "hello";          string s2 = "bye";          Swap<string>(ref s1, ref s2);          Console.WriteLine($"s1={s1}    s2={s2}"); // s1=bye   s2=hello            Console.Read();      }      public static void Swap<T> (ref T x, ref T y)      {          T temp = x;          x = y;          y = temp;      }  } |

# Ограничения методов

Подобным образом можно использовать и ограничения методов:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19 | private static void Main(string[] args)  {      Account<int> acc1 = new Account<int>(1857) { Sum = 4500 };      Account<int> acc2 = new Account<int>(3453) { Sum = 5000 };        Transact<Account<int>>(acc1, acc2, 900);        Console.Read();  }    public static void Transact<T>(T acc1, T acc2, int sum) where T : Account<int>  {      if (acc1.Sum > sum)      {          acc1.Sum -= sum;          acc2.Sum += sum;      }      Console.WriteLine($"acc1: {acc1.Sum}   acc2: {acc2.Sum}");  } |

# Преобразование типов Nullable

Рассмотрим возможные преобразования:

* явное преобразование от T? к T

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | int? x1 = null;  if(x1.HasValue)  {      int x2 = (int)x1;      Console.WriteLine(x2);  } |

* неявное преобразование от T к T?

|  |  |
| --- | --- |
| 1  2  3 | int x1 = 4;  int? x2 = x1;  Console.WriteLine(x2); |

* неявные расширяющие преобразования от V к T?

|  |  |
| --- | --- |
| 1  2  3 | int x1 = 4;  long? x2 = x1;  Console.WriteLine(x2); |

* явные сужающие преобразования от V к T?

|  |  |
| --- | --- |
| 1  2 | long x1 = 4;  int? x2 = (int?)x1; |

* Подобным образом работают преобразования от V? к T?

|  |  |
| --- | --- |
| 1  2 | long? x1 = 4;  int? x2 = (int?)x1; |

* явные преобразования от V? к T

|  |  |
| --- | --- |
| 1  2 | long? x1 = 4;  int x2 = (int)x1; |

### Оператор ??

Оператор ?? называется оператором **null-объединения**. Он применяется для установки значений по умолчанию для типов значений и ссылочных типов, которые допускают значение null. Оператор ?? возвращает левый операнд, если этот операнд не равен null. Иначе возвращается правый операнд. При этом левый операнд должен принимать null. Посмотрим на примере:

|  |  |
| --- | --- |
| 1  2  3  4  5 | int? x = null;  int y = x ?? 100;  // равно 100, так как x равен null    int? z = 200;  int t = z ?? 44; // равно 200, так как z не равен null |

Но мы не можем написать следующим образом:

|  |  |
| --- | --- |
| 1  2 | int x = 44;  int y = x ?? 100; |

Здесь переменная x не является nullable-типом и не может принимать значение null, поэтому в качестве левого операнда в операции ?? она использоваться не может.

# Интерфейсы

У интерфейса методы и свойства не имеют реализации, в этом они сближаются с абстрактными методами абстрактных классов. Сущность данного интерфейса проста: он определяет два свойства для текущей суммы денег на счете и ставки процента по вкладам и два метода для добавления денег на счет и изъятия денег.

Еще один момент в объявлении интерфейса: все его члены - методы и свойства не имеют модификаторов доступа, но фактически по умолчанию доступ **public**, так как цель интерфейса - определение функционала для реализации его классом. Поэтому весь функционал должен быть открыт для реализации.

### Обобщенные интерфейсы

Как и классы, интерфейсы могут быть обобщенными:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18 | interface IAccount<T>  {      void SetSum(T \_sum);      void Display();  }    class Client<T> : IAccount<T>  {      T sum=default(T);      public void SetSum(T \_sum)      {          this.sum = \_sum;      }      public void Display()      {          Console.WriteLine(sum);      }  } |

Класс Tester реализует интерфейсы IFoo и IBar:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19 | interface IFoo  {      void Execute();  }  interface IBar  {      void Execute();  }  class Tester : IFoo, IBar  {      void IFoo.Execute()      {          Console.WriteLine("IFoo Executes");      }      void IBar.Execute()      {          Console.WriteLine("IBar Executes");      }  } |

Реализация метода Execute() из какого интерфейса будет вызвана в следующем коде:

|  |  |
| --- | --- |
| 1  2 | Tester t = new Tester();  t.Execute(); |

Ответ

Выше приведенный код не будет скомпилирован. Необходимо явно указать, реализацию какого интерфейса мы хотим использовать, например, с помощью операции приведения типов:

|  |  |
| --- | --- |
| 1  2 | Tester t = new Tester();  (t as IFoo).Execute(); |

# IClonable

Для сокращения кода копирования мы можем использовать специальный метод **MemberwiseClone()**, который возвращает копию объекта:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | class Person : ICloneable  {      public string Name { get; set; }      public int Age { get; set; }      public object Clone()      {          return this.MemberwiseClone();      }  } |

Этот метод реализует **поверхностное (неглубокое) копирование**. Однако данного копирования может быть недостаточно. Например, пусть класс Person содержит ссылку на объект Company:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16 | class Person : ICloneable  {      public string Name { get; set; }      public int Age { get; set; }      public Company Work { get; set; }        public object Clone()      {              return this.MemberwiseClone();      }  }    class Company  {      public string Name { get; set; }  } |

В этом случае при копировании новая копия будет указывать на тот же объект Company:

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | Person p1 = new Person { Name="Tom", Age = 23, Work= new Company { Name = "Microsoft" } };  Person p2 = (Person)p1.Clone();  p2.Work.Name = "Google";  p2.Name = "Alice";  Console.WriteLine(p1.Name); // Tom  Console.WriteLine(p1.Work.Name); // Google - а должно быть Microsoft |

Поверхностное копирование работает только для свойств, представляющих примитивные типы, но не для сложных объектов. И в этом случае надо применять **глубокое копирование**:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22 | class Person : ICloneable  {      public string Name { get; set; }      public int Age { get; set; }      public Company Work { get; set; }        public object Clone()      {          Company company = new Company { Name = this.Work.Name };          return new Person          {              Name = this.Name,              Age = this.Age,              Work = company          };      }  }    class Company  {      public string Name { get; set; } |

# IComparer

Создадим компаратор объектов Person. Пусть он сравнивает объекты в зависимости от длины строки - значения свойства Name:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12 | class PeopleComparer : IComparer<Person>  {      public int Compare(Person p1, Person p2)      {          if (p1.Name.Length > p2.Name.Length)              return 1;          else if (p1.Name.Length < p2.Name.Length)              return -1;          else              return 0;      }  } |

# Делегаты

Также делегаты могут быть параметрами методов:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29 | class Program  {      delegate void GetMessage();        static void Main(string[] args)      {          if (DateTime.Now.Hour < 12)          {              Show\_Message(GoodMorning);          }          else          {               Show\_Message(GoodEvening);          }          Console.ReadLine();      }      private static void Show\_Message(GetMessage \_del)      {          \_del.Invoke();      }      private static void GoodMorning()      {          Console.WriteLine("Good Morning");      }      private static void GoodEvening()      {          Console.WriteLine("Good Evening");      } |

# Функционал методов обратного вызова

class Account

{

    // Объявляем делегат

    public delegate void AccountStateHandler(string message);

    // Создаем переменную делегата

    AccountStateHandler del;

    // Регистрируем делегат

    public void RegisterHandler(AccountStateHandler \_del)

    {

        del = \_del;

    }

public void Withdraw(int sum)

{

    if (sum <= \_sum)

    {

         \_sum -= sum;

        if (del != null)

            del("Сумма " + sum.ToString() + " снята со счета");

    }

    else

    {

        if (del != null)

            del("Недостаточно денег на счете");

    }

}

class Program

{

    static void Main(string[] args)

    {

        // создаем банковский счет

        Account account = new Account(200, 6);

        // Добавляем в делегат ссылку на метод Show\_Message

        // а сам делегат передается в качестве параметра метода RegisterHandler

        account.RegisterHandler(new Account.AccountStateHandler(Show\_Message));

        // Два раза подряд пытаемся снять деньги

        account.Withdraw(100);

        account.Withdraw(150);

        Console.ReadLine();

    }

    private static void Show\_Message(String message)

    {

        Console.WriteLine(message);

    }

}

Опять же может возникнуть вопрос: почему бы к коде метода Withdraw() не выводить сообщение о снятии денег? Зачем нужно задействовать какой-то делегат?

Дело в том, что не всегда у нас есть доступ к коду классов. Например, часть классов может создаваться и компилироваться одним человеком, который не будет знать, как эти классы будут использоваться. А использовать эти классы будет другой разработчик.

Так, здесь мы выводим сообщение на консоль. Однако для класса Account не важно, как это сообщение выводится. Классу Account даже не известно, что вообще будет делаться в результате списания денег. Он просто посылает уведомление об этом через делегат.

В результате, если мы создаем консольное приложение, мы можем через делегат выводить сообщение на консоль. Если мы создаем графическое приложение Windows Forms или WPF, то можно выводить сообщение в виде графического окна. А можно не просто выводить сообщение. А, например, записать при списании информацию об этом действии в файл или отправить уведомление на электронную почту. В общем любыми способами обработать вызов делегата. И способ обработки не будет зависеть от класса Account.

# События

class Account

{

    // Объявляем делегат

    public delegate void AccountStateHandler(string message);

    // Событие, возникающее при выводе денег

    public event AccountStateHandler Withdrowed;

    // Событие, возникающее при добавление на счет

    public event AccountStateHandler Added;

    int \_sum; // Переменная для хранения суммы

    int \_percentage; // Переменная для хранения процента

    public Account(int sum, int percentage)

    {

        \_sum = sum;

        \_percentage = percentage;

    }

    public int CurrentSum

    {

        get { return \_sum; }

    }

    public void Put(int sum)

    {

        \_sum += sum;

        if (Added != null)

            Added("На счет поступило " + sum);

    }

    public void Withdraw(int sum)

    {

        if (sum <= \_sum)

        {

            \_sum -= sum;

            if (Withdrowed != null)

                Withdrowed("Сумма " + sum + " снята со счета");

        }

        else

        {

            if (Withdrowed != null)

                Withdrowed("Недостаточно денег на счете");

        }

    }

    public int Percentage

    {

        get { return \_percentage; }

    }

}

class Program

{

    static void Main(string[] args)

    {

        Account account = new Account(200, 6);

        // Добавляем обработчики события

        account.Added += Show\_Message;

        account.Withdrowed += Show\_Message;

        account.Withdraw(100);

        // Удаляем обработчик события

        account.Withdrowed -= Show\_Message;

        account.Withdraw(50);

        account.Put(150);

        Console.ReadLine();

    }

    private static void Show\_Message(string message)

    {

        Console.WriteLine(message);

    }

}

# EventArgs

class AccountEventArgs

{

    // Сообщение

    public string message;

    // Сумма, на которую изменился счет

    public int sum;

    public AccountEventArgs(string \_mes, int \_sum)

    {

        message = \_mes;

        sum = \_sum;

    }

}

class Account

{

    // Объявляем делегат

    public delegate void AccountStateHandler(object sender, AccountEventArgs e);

    // Событие, возникающее при выводе денег

    public event AccountStateHandler Withdrowed;

    // Событие, возникающее при добавлении на счет

    public event AccountStateHandler Added;

    int \_sum; // Переменная для хранения суммы

    int \_percentage; // Переменная для хранения процента

    public Account(int sum, int percentage)

    {

        \_sum = sum;

        \_percentage = percentage;

    }

    public int CurrentSum

    {

        get { return \_sum; }

    }

    public void Put(int sum)

    {

        \_sum += sum;

        if (Added != null)

            Added(this, new AccountEventArgs("На счет поступило " + sum, sum));

    }

    public void Withdraw(int sum)

    {

        if (sum <= \_sum)

        {

            \_sum -= sum;

            if (Withdrowed != null)

                Withdrowed(this, new AccountEventArgs("Сумма " + sum + " снята со счета", sum));

        }

        else

        {

            if (Withdrowed != null)

                Withdrowed(this, new AccountEventArgs("Недостаточно денег на счете", sum));

        }

    }

    public int Percentage

    {

        get { return \_percentage; }

    }

class Program

{

    static void Main(string[] args)

    {

        Account account = new Account(200, 6);

        // Добавляем обработчики события

        account.Added += Show\_Message;

        account.Withdrowed += Show\_Message;

        account.Withdraw(100);

        // Удаляем обработчик события

        account.Withdrowed -= Show\_Message;

        account.Withdraw(50);

        account.Put(150);

        Console.ReadLine();

    }

    private static void Show\_Message(object sender, AccountEventArgs e)

    {

        Console.WriteLine("Сумма транзакции: {0}", e.sum);

        Console.WriteLine(e.message);

    }

}

# Анонимные методы

Account account = new Account(200, 6);

// Добавляем обработчики события

account.Added += delegate(object sender, AccountEventArgs e)

{

    Console.WriteLine("Сумма транзакции: {0}", e.sum);

    Console.WriteLine(e.message);

};

delegate void GetMessage();

static void Main(string[] args)

{

    GetMessage message = delegate

    {

        Console.WriteLine("анонимный делегат");

    };

    message();

    Console.Read();

}

# Лямбды

class Program

{

    delegate int Square(int x); // объявляем делегат, принимающий int и возвращающий int

    static void Main(string[] args)

    {

        Square squareInt = i => i \* i; // объекту делегата присваивается лямбда-выражение

        int z = squareInt(6); // используем делегат

        Console.WriteLine(z); // выводит число 36

        Console.Read();

    }

}

Account account = new Account(200, 6);

account.Added += (sender, e)=>

{

    Console.WriteLine("Сумма транзакции: {0}", e.sum);

    Console.WriteLine(e.message);

};

class Program

{

    delegate void message(); // делегат без параметров

    static void Main(string[] args)

    {

        message GetMessage = () => { Console.WriteLine("Лямбда-выражение"); };

        GetMessage();

        Console.Read();

    }

}

class Program

{

    delegate void message(); // делегат без параметров

    static void Main(string[] args)

    {

        message GetMessage = () => Show\_Message();

        GetMessage();

    }

    private static void Show\_Message()

    {

        Console.WriteLine("Привет мир!");

    }

}

class Program

{

    delegate bool IsEqual(int x);

    static void Main(string[] args)

    {

        int[] integers = { 1, 2, 3, 4, 5, 6, 7, 8, 9 };

        // найдем сумму чисел больше 5

        int result1 = Sum(integers, x => x > 5);

        Console.WriteLine(result1); // 30

        // найдем сумму четных чисел

        int result2 = Sum(integers, x => x % 2 == 0);

        Console.WriteLine(result2);  //20

        Console.Read();

    }

    private static int Sum (int[] numbers, IsEqual func)

    {

        int result = 0;

        foreach(int i in numbers)

        {

            if (func(i))

                result += i;

        }

        return result;

    }

}

# Ковариантность и контравариантность делегатов

Благодаря ковариантности мы можем присвоить делегату метод, возвращаемым типом которого является тип, производный от того типа, который возвращается делегатом

class Person

{

    public string Name { get; set; }

    public Person(string name)

    {

        Name = name;

    }

    public void Display()

    {

        Console.WriteLine(Name);

    }

}

class Client : Person

{

    public Client(string name) : base(name)

    {

    }

}

delegate Person PersonFactory(string name);

static void Main(string[] args)

{

    PersonFactory personDel;

    personDel = BuildClient; // ковариантность

    Person p = personDel("Tom");

    p.Display();

    Console.Read();

}

private static Client BuildClient(string name)

{

    return new Client(name);

}

Благодаря контравариантности можно присвоить делегату метод, тип параметра которого представляет базовый класс по отношению к классу параметра, который используется делегатом.

delegate void ClientInfo(Client client);

static void Main(string[] args)

{

    ClientInfo clientInfo = GetPersonInfo; // контравариантность

    Client client = new Client("Alice");

    clientInfo(client);

    Console.Read();

}

private static void GetPersonInfo(Person p)

{

    p.Display();

}

# Ковариантность и контравариантность в обобщенных делегатах

class Person

{

    public string Name { get; set; }

    public Person(string name)

    {

        Name = name;

    }

    public virtual void Display()

    {

        Console.WriteLine("Person: " +Name);

    }

}

class Client : Person

{

    public Client(string name) : base(name)

    { }

    public override void Display()

    {

        Console.WriteLine("Client: " + Name);

    }

}

Теперь объявим ковариантный обобщенный делегат

class Program

{

    delegate T Builder<out T>(string name);

    static void Main(string[] args)

    {

        Builder<Person> personBuilder = GetPerson;

        Builder<Client> clientBuilder = GetClient;

        personBuilder = clientBuilder; // ковариантность

        Person p = personBuilder("Tom"); // вызов делегата

        p.Display(); // Client: Tom

        Console.Read();

    }

    private static Person GetPerson(string name)

    {

        return new Person(name);

    }

    private static Client GetClient(string name)

    {

        return new Client(name);

    }

}

Рассмотрим контравариантный делегат

class Program

{

    delegate void GetInfo<in T>(T item);

    static void Main(string[] args)

    {

        GetInfo<Person> personInfo = PersonInfo;

        GetInfo<Client> clientInfo = ClientInfo;

        clientInfo = personInfo; // контравариантность

        Client client = new Client("Tom");

        clientInfo(client); // Client: Tom

        Console.Read();

    }

    private static void PersonInfo(Person p)

    {

        p.Display();

    }

    private static void ClientInfo(Client cl)

    {

        cl.Display();

    }

}

# Делегаты Action, Predicate и Func

### Action

Делегат Action является обобщенным, принимает параметры и возвращает значение void:

public delegate void Action<T>(T obj)

Данный делегат имеет ряд перегруженных версий. Каждая версия принимает разное число параметров: от Action<in T1> до Action<in T1, in T2,....in T16>. Таким образом можно передать до 16 значений в метод.

static void Main(string[] args)

{

    Action<int, int> op;

    op = Add;

    Operation(10, 6, op);

    op = Substract;

    Operation(10, 6, op);

    Console.Read();

}

static void Operation(int x1, int x2, Action<int, int> op)

{

    if (x1 > x2)

        op(x1, x2);

}

static void Add(int x1, int x2)

{

    Console.WriteLine("Сумма чисел: " + (x1 + x2));

}

### Predicate

Делегат Predicate<T>, как правило, используется для сравнения, сопоставления некоторого объекта T определенному условию. В качестве выходного результата возвращается значение true, если условие соблюдено, и false, если не соблюдено

Predicate<int> isPositive = delegate (int x) { return x > 0; };

Console.WriteLine(isPositive(20));

Console.WriteLine(isPositive(-20));

### Func

Еще одним распространенным делегатом является **Func**. Он возвращает результат действия и может принимать параметры. Он также имеет различные формы: от Func<out T>(), где T - тип возвращаемого значения, до Func<in T1, in T2,...in T16, out TResult>(), то есть может принимать до 16 параметров

static void Main(string[] args)

{

    Func<int, int> retFunc = Factorial;

    int n1 = GetInt(6, retFunc);

    Console.WriteLine(n1);  // 720

    int n2 = GetInt(6, x=> x \*x);

    Console.WriteLine(n2); // 36

    Console.Read();

}

static int GetInt(int x1, Func<int, int> retF)

{

    int result = 0;

    if (x1 > 0)

        result = retF(x1);

    return result;

}

static int Factorial(int x)

{

    int result = 1;

    for (int i = 1; i <= x; i++)

    {

        result \*= i;

    }

    return result;

}

# Пространства имен

Пространства имен могут быть определены внутри других пространств

using HelloApp.AccountSpace;

namespace HelloApp

{

    class Program

    {

        static void Main(string[] args)

        {

            Account account = new Account(4);

        }

    }

    namespace AccountSpace

    {

        class Account

        {

            public int Id { get; private set;}

            public Account(int \_id)

            {

                Id = \_id;

            }

        }

    }

}

### Псевдонимы

Для различных классов мы можем использовать псевдонимы. Затем в программе вместо названия класса используется его псевдоним. Например, для вывода строки на экран применяется метод Console.WriteLine(). Но теперь зададим для класса Console псевдоним

using printer = System.Console;

class Program

{

    static void Main(string[] args)

    {

        printer.WriteLine("Hello from C#");

        printer.Read();

    }

}

И еще пример. Определим класс и для него псевдоним

using Person = HelloApp.User;

using Printer = System.Console;

namespace HelloApp

{

    class Program

    {

        static void Main(string[] args)

        {

            Person person = new Person();

            person.name = "Tom";

            Printer.WriteLine(person.name);

            Printer.Read();

        }

    }

    class User

    {

        public string name;

    }

}

Начиная с версии C# 6.0 (Visual Studio 2015) в язык была добавлена возможность импорта функциональности классов. Например, опять же возьмем класс Console и применим новую функциональность

using static System.Console;

namespace HelloApp

{

    class Program

    {

        static void Main(string[] args)

        {

            WriteLine("Hello from C# 6.0");

            Read();

        }

    }

}

namespace HelloApp

{

    class Program

    {

        static void Main(string[] args)

        {

            double radius = 50;

            double result = GetArea(radius); //Geometry.GetArea

            WriteLine(result); //Console.WriteLine

            Read(); // Console.Read

        }

    }

    class Geometry

    {

        public static double GetArea(double radius)

        {

            return PI \* radius \* radius; // Math.PI

        }

    }

}

# Методы расширения

class Program

{

    static void Main(string[] args)

    {

        string s = "Привет мир";

        char c = 'и';

        int i = s.WordCount(c);

        Console.WriteLine(i);

        Console.ReadLine();

    }

}

public static class StringExtension

{

    public static int WordCount(this string str, char c)

    {

        int counter = 0;

        for (int i = 0; i<str.Length; i++)

        {

            if (str[i] == c)

                counter++;

        }

        return counter;

    }

}

# Встроенные выражения (Expression–Bodied)

Действие каждого метода определяется в виде одного выражения или одной строки. Теперь применим Expression–Bodied методы

class Person

{

    public string Name { get;set;}

    public void Display() => Console.WriteLine(Name);

}

class Circle

{

    public static double GetArea(double radius) => radius \* radius \* 3.14;

}

Также можно применять лямбды для упрощения написания свойств, которые содержат только блок get и которые возвращают определенное значение

class Person

{

    // эквивалентно public string Name { get { return name; } }

    public string Name => name;

    private readonly string name;

    public Person(string \_name)

    {

        name = \_name;

    }

    public void Display() => Console.WriteLine(Name);

}

class Program

{

    static void Main(string[] args)

    {

        Person person = new Person("Tom");

        person.Display();

        Console.Read();

    }

}

# Создание классов исключений и класс Exception

Некоторые свойства класса Exception:

* **Helplink**: хранит адрес интернет-ресурса, на котром можно найти всю информацию об ошибке
* **InnerException**: объект класса Exception, хранит информацию об исключении, которое послужило причиной текущего исключения
* **Message**: хранит сообщение об исключении, текст ошибки
* **Source**: хранит имя объекта или сборки, которое вызвало исключение
* **StackTrace**: возвращает строковое представление стека вызывов, которые привели к возникновению исключения
* **TargetSite**: возвращает метод, в котором и было вызвано исключение
* class Program
* {
* static void Main(string[] args)
* {
* try
* {
* Person p = new Person();
* p.Age = 17;
* }
* catch (Exception ex)
* {
* Console.WriteLine("Ошибка: " + ex.Message);
* Console.WriteLine("Метод: " + ex.TargetSite); // Void set\_Age(Int32)
* }
* Console.ReadLine();
* }
* }
* class Person
* {
* private string name;
* public string Name
* {
* get { return name; }
* set { name = value; }
* }
* private int age;
* public int Age
* {
* get { return age; }
* set
* {
* if (value < 18)
* {
* throw new Exception("Лицам до 18 регистрация запрещена");
* }
* else
* {
* age = value;
* }
* }
* }
* }

# Анонимные типы

Анонимные типы позволяют создать объект с некоторым набором свойств без определения класса. Анонимный тип определяется с помощью ключевого слова **var** и инициализатора объектов

var user = new { Name = "Tom", Age = 34 };

Console.WriteLine(user.Name);

Если в программе используются несколько объектов анонимных типов с одинаковым набором свойств, то для них компилятор создаст одно определение анонимного типа

var user = new { Name = "Tom", Age = 34 };

var student = new { Name = "Alice", Age = 21 };

var manager = new { Name = "Bob", Age = 26, Company = "Microsoft" };

Console.WriteLine(user.GetType().Name); // <>f\_\_AnonymousType0'2

Console.WriteLine(student.GetType().Name); // <>f\_\_AnonymousType0'2

Console.WriteLine(manager.GetType().Name); // <>f\_\_AnonymousType1'3

# Локальные функции

class Program

{

    static void Main(string[] args)

    {

        var result = GetResult(new int[] { -3, -2, -1, 0, 1, 2, 3 });

        Console.WriteLine(result);  // 6

        Console.Read();

    }

    static int GetResult(int[] numbers)

    {

        int limit = 0;

        // локальная функция

        bool IsMoreThan(int number)

        {

            return number > limit;

        }

        int result = 0;

        for(int i=0; i < numbers.Length; i++)

        {

            if (IsMoreThan(numbers[i]))

            {

                result += numbers[i];

            }

        }

        return result;

    }

}

# Pattern matching

class Employee

{

    public virtual void Work()

    {

        Console.WriteLine("Да работаю я, работаю");

    }

}

class Manager : Employee

{

    public override void Work()

    {

        Console.WriteLine("Отлично, работаем дальше");

    }

    public bool IsOnVacation { get; set; }

}

Несмотря на то, что любой из трех случаев довольно прост в использовании, однако функциональность **pattern matching** позволяет нам сократить объем кода

static void Main(string[] args)

{

    Employee emp = new Manager(); //Employee();

    UseEmployee(emp);

    Console.Read();

}

static void UseEmployee( Employee emp)

{

    if (emp is Manager manager && manager.IsOnVacation==false)

    {

        manager.Work();

    }

    else

    {

        Console.WriteLine("Преобразование не допустимо");

    }

}

static void UseEmployee( Employee emp)

{

    switch (emp)

    {

        case Manager manager when manager.IsOnVacation==false:

            manager.Work();

            break;

        case null:

            Console.WriteLine("Объект не задан");

            break;

        default:

            Console.WriteLine("Объект не менеджер");

            break;

    }

}

# Деконструкторы

Деконструкторы (не путать с деструкторами) позволяют выполнить декомпозицию объекта на отдельные части. Деконструкторы доступны, начиная с версии C# 7.0 (Visual Studio 2017)

class Person

{

    public string Name { get; set; }

    public int Age { get; set; }

    public void Deconstruct(out string name, out int age)

    {

        name = this.Name;

        age = this.Age;

    }

}

В этом случае мы могли бы выполнить декомпозицию объекта Person так

Person person = new Person { Name = "Tom", Age = 33 };

(string name, int age) = person;

Console.WriteLine(name);    // Tom

Console.WriteLine(age);     // 33

По сути деконструкторы это не более,чем синтаксический сахар. Это все равно, что если бы мы написали в предыдущих версиях C# следующий набор выражений

Person person = new Person { Name = "Tom", Age = 33 };

string name; int age;

person.Deconstruct(out name, out age);

# Оператор условного null

Начиная с C# 6.0 в языке появился элвис-оператор или по другому оператор условного null (Null-Conditional Operator). Он позволяет упростить проверку объектов на значение **null** в условных конструкциях

class User

{

    public Phone Phone { get; set; }

}

class Phone

{

    public Company Company { get; set; }

}

class Company

{

    public string Name { get; set; }

}

User user = new User();

user.Phone=new Phone { Company = new Company { Name = "Samsung" } };

string companyName = user.Phone.Company.Name;

if(user!=null && user.Phone!=null && user.Phone.Company!=null)

{

    string companyName = user.Phone.Company.Name;

    Console.WriteLine(companyName);

}

Выражение ?. и представляет элвис-оператор. Здесь последовательно проверяется равен ли объект user и вложенные объекты значению null. Если же на каком-то этапе один из объектов окажется равным null, то companyName будет иметь значение по умолчанию, то есть пустую строку.

И в этом случае мы можем пойти дальше и применить операцию ?? для установки значения по умолчанию, если строка является пустой

User user = new User();

string companyName = user?.Phone?.Company?.Name??"не установлено";

Console.WriteLine(companyName);

# Коллекции

## Необобщенные коллекции

ассмотрим основные интерфейсы:

* **IEnumerable**: определяет метод GetEnumerator. Данный метод возвращает перечислитель - то есть некоторый объект, реализующий интерфейс IEnumerator.
* **IEnumerator**: реализация данного интерфейса позволяет перебирать элементы коллекции с помощью цикла foreach
* **ICollection**: является основой для всех необобщенных коллекций, определяет основные методы и свойства для всех необобщенных коллекций (например, метод CopyTo и свойство Count). Данный интерфейс унаследован от интерфейса IEnumerable, благодаря чему базовый интерфейс также реализуется всеми классами необобщенных коллекций
* **IList**: позволяет получать элементы коллекции по порядку. Также определяет ряд методов для манипуляции элементами: Add (добавление элементов), Remove/RemoveAt (удаление элемента) и ряд других.
* **IComparer**: определяет метод int Compare(object x, object y) для сравнения двух объектов
* **IDictionary**: определяет поведение коллекции, при котором она должна хранить объекты в виде пар ключ-значение: для каждого объекта определяется уникальный ключ, и этому ключу соответствует определенное значение
* **IDictionaryEnumerator**: определяет методы и свойства для перечислителя словаря
* **IEqualityComparer**: определяет два метода Equals и GetHashCode, с помощью которых два объекта сравниваются на предмет равенства
* **IStructuralComparer**: определяет метод Compare для структурного сравнения двух объектов: при таком сравнении сравниваются не ссылки на объекты, а непосредственное содержимое объектов
* **IStructuralEquatable**: позволяет провести структурное равенство двух объектов. Как и в случае с интерфейсом IStructuralComparer сравнивается содержимое двух объектов

Эти интерфейсы реализуются следующими классами коллекций в пространстве имен System.Collections:

* **ArrayList**: класс простой коллекции объектов. Реализует интерфейсы IList, ICollection, IEnumerable
* **BitArray**: класс коллекции, содержащей массив битовых значений. Реализует интерфейсы ICollection, IEnumerable
* **Hashtable**: класс коллекции, представляющей хэш-таблицу и храняющий набор пар "ключ-значение"
* **Queue**: класс очереди объектов, работающей по алгоритму FIFO("первый вошел -первый вышел"). Реализует интерфейсы ICollection, IEnumerable
* **SortedList**: класс коллекции, хранящей наборы пар "ключ-значение", отсортированных по ключу. Реализует интерфейсы ICollection, IDictionary, IEnumerable
* **Stack**: класс стека

# Список List<T>

Среди его методов можно выделить следующие:

* **void Add(T item)**: добавление нового элемента в список
* **void AddRange(ICollection collection)**: добавление с список коллекции или массива
* **int BinarySearch(T item)**: бинарный поиск элемента в списке. Если элемент найден, то метод возвращает индекс этого элемента в коллекции. При этом список должен быть отсортирован.
* **int IndexOf(T item)**: возвращает индекс первого вхождения элемента в списке
* **void Insert(int index, T item)**: вставляет элемент item в списке на позицию index
* **bool Remove(T item)**: удаляет элемент item из списка, и если удаление прошло успешно, то возвращает true
* **void RemoveAt(int index)**: удаление элемента по указанному индексу index
* **void Sort()**: сортировка списка

# Двухсвязный список LinkedList<T>

сли в простом списке List<T> каждый элемент представляет объект типа T, то в LinkedList<T> каждый узел представляет объект класса LinkedListNode<T>. Этот класс имеет следующие свойства:

* **Value**: само значение узла, представленное типом T
* **Next**: ссылка на следующий элемент типа LinkedListNode<T> в списке. Если следующий элемент отсутствует, то имеет значение null
* **Previous**: ссылка на предыдущий элемент типа LinkedListNode<T> в списке. Если предыдущий элемент отсутствует, то имеет значение null

Используя методы класса LinkedList<T>, можно обращаться к различным элементам, как в конце, так и в начале списка:

* **AddAfter(LinkedListNode<T> node, LinkedListNode<T> newNode)**: вставляет узел newNode в список после узла node.
* **AddAfter(LinkedListNode<T> node, T value)**: вставляет в список новый узел со значением value после узла node.
* **AddBefore(LinkedListNode<T> node, LinkedListNode<T> newNode)**: вставляет в список узел newNode перед узлом node.
* **AddBefore(LinkedListNode<T> node, T value)**: вставляет в список новый узел со значением value перед узлом node.
* **AddFirst(LinkedListNode<T> node)**: вставляет новый узел в начало списка
* **AddFirst(T value)**: вставляет новый узел со значением value в начало списка
* **AddLast(LinkedListNode<T> node)**: вставляет новый узел в конец списка
* **AddLast(T value)**: вставляет новый узел со значением value в конец списка
* **RemoveFirst()**: удаляет первый узел из списка. После этого новым первым узлом становится узел, следующий за удаленным
* **RemoveLast()**: удаляет последний узел из списка

# Инициализация словарей

В C# 5.0 мы могли инициализировать словари следующим образом:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10 | Dictionary<string, string> countries = new Dictionary<string, string>  {      {"Франция", "Париж"},      {"Германия", "Берлин"},      {"Великобритания", "Лондон"}  };    foreach(var pair in countries)      Console.WriteLine("{0} - {1}", pair.Key, pair.Value); |

То начиная с C# 6.0 (Visual Studio 2015) доступен также еще один способ инициализации:

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | Dictionary<string, string> countries = new Dictionary<string, string>  {      ["Франция"]= "Париж",      ["Германия"]= "Берлин",      ["Великобритания"]= "Лондон"  }; |

# Класс ObservableCollection

class Program

    {

        static void Main(string[] args)

        {

            ObservableCollection<User> users = new ObservableCollection<User>

            {

                new User { Name = "Bill"},

                new User { Name = "Tom"},

                new User { Name = "Alice"}

            };

            users.CollectionChanged += Users\_CollectionChanged;

            users.Add(new User { Name = "Bob" });

            users.RemoveAt(1);

            users[0] = new User{ Name = "Anders" };

            foreach(User user in users)

            {

                Console.WriteLine(user.Name);

            }

            Console.Read();

        }

        private static void Users\_CollectionChanged(object sender, NotifyCollectionChangedEventArgs e)

        {

            switch(e.Action)

            {

                case NotifyCollectionChangedAction.Add: // если добавление

                    User newUser = e.NewItems[0] as User;

                    Console.WriteLine("Добавлен новый объект: {0}", newUser.Name);

                    break;

                case NotifyCollectionChangedAction.Remove: // если удаление

                    User oldUser = e.OldItems[0] as User;

                    Console.WriteLine("Удален объект: {0}", oldUser.Name);

                    break;

                case NotifyCollectionChangedAction.Replace: // если замена

                    User replacedUser = e.OldItems[0] as User;

                    User replacingUser = e.NewItems[0] as User;

                    Console.WriteLine("Объект {0} заменен объектом {1}",

                                        replacedUser.Name, replacingUser.Name);

                    break;

            }

        }

    }

    class User

    {

        public string Name { get; set; }

    }

# Индексаторы и создание коллекций

class Library

{

    Book[] books;

    public Library()

    {

        books = new Book[] { new Book("Отцы и дети"), new Book("Война и мир"),

                new Book("Евгений Онегин") };

    }

    public int Length

    {

        get { return books.Length; }

    }

    public Book this[int index]

    {

        get

        {

            return books[index];

        }

        set

        {

            books[index] = value;

        }

    }

}

Посмотрим более сложный пример. Допустим, у нас есть класс, в котором хранилище определено в виде двухмерного массива или матрицы

class Matrix

{

    private int[,] numbers = new int[,] { { 1, 2, 4}, { 2, 3, 6 }, { 3, 4, 8 } };

    public int this[int i, int j]

    {

        get

        {

            return numbers[i,j];

        }

        set

        {

            numbers[i, j] = value;

        }

    }

}

# Интерфейсы IEnumerable и IEnumerator

class Book

{

    public Book(string name)

    {

        this.Name=name;

    }

    public string Name { get; set; }

}

class Library : IEnumerable

{

    private Book[] books;

    public Library()

    {

        books = new Book[] { new Book("Отцы и дети"), new Book("Война и мир"),

                new Book("Евгений Онегин") };

    }

    public int Length

    {

        get { return books.Length; }

    }

    public Book this[int index]

    {

        get

        {

            return books[index];

        }

        set

        {

            books[index] = value;

        }

    }

    // возвращаем перечислитель

    IEnumerator IEnumerable.GetEnumerator()

    {

        return books.GetEnumerator();

    }

}

class Program

{

    static void Main(string[] args)

    {

        Library library = new Library();

        foreach (Book b in library)

        {

            Console.WriteLine(b.Name);

        }

        Console.ReadLine();

    }

}

# Итераторы и оператор yield

### Именованный итератор

Выше для создания итератора мы использовали метод GetEnumerator. Но оператор yield можно использовать внутри любого метода, только такой метод должен возвращать объект интерфейса IEnumerable. Подобные методы еще называют **именованными итераторами**.

Создадим такой именованный итератор в классе Library и используем его:

class Book

{

    public Book(string name)

    {

        this.Name=name;

    }

    public string Name { get; set; }

}

class Library

{

    private Book[] books;

    public Library()

    {

        books = new Book[] { new Book("Отцы и дети"), new Book("Война и мир"),

            new Book("Евгений Онегин") };

    }

    public int Length

    {

        get { return books.Length; }

    }

    public Book this[int index]

    {

        get

        {

            return books[index];

        }

        set

        {

            books[index] = value;

        }

    }

    public IEnumerable GetBooks(int max)

    {

        for (int i = 0; i < max; i++)

        {

            if (i == books.Length)

            {

                yield break;

            }

            else

            {

                yield return books[i];

            }

        }

    }

}

Применение итератора:

Library library = new Library();

foreach (Book b in library.GetBooks(5))

{

    Console.WriteLine(b.Name);

}

# Работа с дисками

Этот класс имеет статический метод GetDrives, который возвращает имена всех логических дисков компьютера. Также он предоставляет ряд полезных свойств:

* **AvailableFreeSpace**: указывает на объем доступного свободного места на диске в байтах
* **DriveFormat**: получает имя файловой системы
* **DriveType**: представляет тип диска
* **IsReady**: готов ли диск (например, DVD-диск может быть не вставлен в дисковод)
* **Name**: получает имя диска
* **TotalFreeSpace**: получает общий объем свободного места на диске в байтах
* **TotalSize**: общий размер диска в байтах
* **VolumeLabel**: получает или устанавливает метку тома

### Класс Directory

Класс Directory предоставляет ряд статических методов для управления каталогами. Некоторые из этих методов:

* **CreateDirectory(path)**: создает каталог по указанному пути path
* **Delete(path)**: удаляет каталог по указанному пути path
* **Exists(path)**: определяет, существует ли каталог по указанному пути path. Если существует, возвращается true, если не существует, то false
* **GetDirectories(path)**: получает список каталогов в каталоге path
* **GetFiles(path)**: получает список файлов в каталоге path
* **Move(sourceDirName, destDirName)**: перемещает каталог
* **GetParent(path)**: получение родительского каталога

### Класс DirectoryInfo

Данный класс предоставляет функциональность для создания, удаления, перемещения и других операций с каталогами. Во многом он похож на Directory. Некоторые из его свойств и методов:

* **Create()**: создает каталог
* **CreateSubdirectory(path)**: создает подкаталог по указанному пути path
* **Delete()**: удаляет каталог
* Свойство **Exists**: определяет, существует ли каталог
* **GetDirectories()**: получает список каталогов
* **GetFiles()**: получает список файлов
* **MoveTo(destDirName)**: перемещает каталог
* Свойство **Parent**: получение родительского каталога
* Свойство **Root**: получение корневого каталога

Посмотрим на примерах применение этих классов

# Классы File и FileInfo

одобно паре Directory/DirectoryInfo для работы с файлами предназначена пара классов **File** и**FileInfo**. С их помощью мы можем создавать, удалять, перемещать файлы, получать их свойства и многое другое.

Некоторые полезные методы и свойства класса FileInfo:

* **CopyTo(path)**: копирует файл в новое место по указанному пути path
* **Create()**: создает файл
* **Delete()**: удаляет файл
* **MoveTo(destFileName)**: перемещает файл в новое место
* Свойство **Directory**: получает родительский каталог в виде объекта DirectoryInfo
* Свойство **DirectoryName**: получает полный путь к родительскому каталогу
* Свойство **Exists**: указывает, существует ли файл
* Свойство **Length**: получает размер файла
* Свойство **Extension**: получает расширение файла
* Свойство **Name**: получает имя файла
* Свойство **FullName**: получает полное имя файла

Класс File реализует похожую функциональность с помощью статических методов:

* **Copy()**: копирует файл в новое место
* **Create()**: создает файл
* **Delete()**: удаляет файл
* **Move**: перемещает файл в новое место
* **Exists(file)**: определяет, существует ли файл

# Класс FileStream

Класс **FileStream** представляет возможности по считыванию из файла и записи в файл. Он позволяет работать как с текстовыми файлами, так и с бинарными.

Рассмотрим наиболее важные его свойства и методы:

* Свойство **Length**: возвращает длину потока в байтах
* Свойство **Position**: возвращает текущую позицию в потоке
* Метод **Read**: считывает данные из файла в массив байтов. Принимает три параметра: int Read(byte[] array, int offset, int count) и возвращает количество успешно считанных байтов. Здесь используются следующие параметры:
  + array - массив байтов, куда будут помещены считываемые из файла данные
  + offset представляет смещение в байтах в массиве array, в который считанные байты будут помещены
  + count - максимальное число байтов, предназначенных для чтения. Если в файле находится меньшее количество байтов, то все они будут считаны.
* Метод **long Seek(long offset, SeekOrigin origin)**: устанавливает позицию в потоке со смещением на количество байт, указанных в параметре offset.
* Метод **Write**: записывает в файл данные из массива байтов. Принимает три параметра:Write(byte[] array, int offset, int count)
  + array - массив байтов, откуда данные будут записываться в файла
  + offset - смещение в байтах в массиве array, откуда начинается запись байтов в поток
  + count - максимальное число байтов, предназначенных для записи
* Console.WriteLine("Введите строку для записи в файл:");
* string text = Console.ReadLine();
* // запись в файл
* using (FileStream fstream = new FileStream(@"C:\SomeDir\noname\note.txt", FileMode.OpenOrCreate))
* {
* // преобразуем строку в байты
* byte[] array = System.Text.Encoding.Default.GetBytes(text);
* // запись массива байтов в файл
* fstream.Write(array, 0, array.Length);
* Console.WriteLine("Текст записан в файл");
* }
* // чтение из файла
* using (FileStream fstream = File.OpenRead(@"C:\SomeDir\noname\note.txt"))
* {
* // преобразуем строку в байты
* byte[] array = new byte[fstream.Length];
* // считываем данные
* fstream.Read(array, 0, array.Length);
* // декодируем байты в строку
* string textFromFile = System.Text.Encoding.Default.GetString(array);
* Console.WriteLine("Текст из файла: {0}", textFromFile);
* }
* Console.ReadLine();
* **Append**: если файл существует, то текст добавляется в конец файл. Если файла нет, то он создается. Файл открывается только для записи.
* **Create**: создается новый файл. Если такой файл уже существует, то он перезаписывается
* **CreateNew**: создается новый файл. Если такой файл уже существует, то он приложение выбрасывает ошибку
* **Open**: открывает файл. Если файл не существует, выбрасывается исключение
* **OpenOrCreate**: если файл существует, он открывается, если нет - создается новый
* **Truncate**: если файл существует, то он перезаписывается. Файл открывается только для записи.
* using System.IO;
* using System.Text;
* class Program
* {
* static void Main(string[] args)
* {
* string text = "hello world";
* // запись в файл
* using (FileStream fstream = new FileStream(@"D:\note.dat", FileMode.OpenOrCreate))
* {
* // преобразуем строку в байты
* byte[] input = Encoding.Default.GetBytes(text);
* // запись массива байтов в файл
* fstream.Write(input, 0, input.Length);
* Console.WriteLine("Текст записан в файл");
* // перемещаем указатель в конец файла, до конца файла- пять байт
* fstream.Seek(-5, SeekOrigin.End); // минус 5 символов с конца потока
* // считываем четыре символов с текущей позиции
* byte[] output = new byte[4];
* fstream.Read(output, 0, output.Length);
* // декодируем байты в строку
* string textFromFile = Encoding.Default.GetString(output);
* Console.WriteLine("Текст из файла: {0}", textFromFile); // worl
* // заменим в файле слово world на слово house
* string replaceText = "house";
* fstream.Seek(-5, SeekOrigin.End); // минус 5 символов с конца потока
* input = Encoding.Default.GetBytes(replaceText);
* fstream.Write(input, 0, input.Length);
* // считываем весь файл
* // возвращаем указатель в начало файла
* fstream.Seek(0, SeekOrigin.Begin);
* output = new byte[fstream.Length];
* fstream.Read(output, 0, output.Length);
* // декодируем байты в строку
* textFromFile = Encoding.Default.GetString(output);
* Console.WriteLine("Текст из файла: {0}", textFromFile); // hello house
* }
* Console.Read();
* }

FileStream fstream = null;

try

{

    fstream = new FileStream(@"D:\note3.dat", FileMode.OpenOrCreate);

    // операции с потоком

}

catch(Exception ex)

{

}

finally

{

    if (fstream != null)

        fstream.Close();

}

# StreamReader и StreamWriter

### Чтение из файла и StreamReader

Класс StreamReader позволяет нам легко считывать весь текст или отдельные строки из текстового файла. Среди его методов можно выделить следующие:

* **Close**: закрывает считываемый файл и освобождает все ресурсы
* **Peek**: возвращает следующий доступный символ, если символов больше нет, то возвращает -1
* **Read**: считывает и возвращает следующий символ в численном представлении. Имеет перегруженную версию: Read(char[] array, int index, int count), где array - массив, куда считываются символы, index - индекс в массиве array, начиная с которого записываются считываемые символы, и count - максимальное количество считываемых символов
* **ReadLine**: считывает одну строку в файле
* **ReadToEnd**: считывает весь текст из файла
* string path= @"C:\SomeDir\hta.txt";
* try
* {
* Console.WriteLine("\*\*\*\*\*\*считываем весь файл\*\*\*\*\*\*\*\*");
* using (StreamReader sr = new StreamReader(path))
* {
* Console.WriteLine(sr.ReadToEnd());
* }
* Console.WriteLine();
* Console.WriteLine("\*\*\*\*\*\*считываем построчно\*\*\*\*\*\*\*\*");
* using (StreamReader sr = new StreamReader(path, System.Text.Encoding.Default))
* {
* string line;
* while ((line = sr.ReadLine()) != null)
* {
* Console.WriteLine(line);
* }
* }
* Console.WriteLine();
* Console.WriteLine("\*\*\*\*\*\*считываем блоками\*\*\*\*\*\*\*\*");
* using (StreamReader sr = new StreamReader(path, System.Text.Encoding.Default))
* {
* char[] array = new char[4];
* // считываем 4 символа
* sr.Read(array, 0, 4);
* Console.WriteLine(array);
* }
* }
* catch (Exception e)
* {
* Console.WriteLine(e.Message);
* }

### Запись в файл и StreamWriter

Для записи в текстовый файл используется класс StreamWriter. Свою функциональность он реализует через следующие методы:

* **Close**: закрывает записываемый файл и освобождает все ресурсы
* **Flush**: записывает в файл оставшиеся в буфере данные и очищает буфер.
* **Write**: записывает в файл данные простейших типов, как int, double, char, string и т.д.
* **WriteLine**: также записывает данные, только после записи добавляет в файл символ окончания строки
* string readPath= @"C:\SomeDir\hta.txt";
* string writePath = @"C:\SomeDir\ath.txt";
* string text = "";
* try
* {
* using (StreamReader sr = new StreamReader(readPath, System.Text.Encoding.Default))
* {
* text=sr.ReadToEnd();
* }
* using (StreamWriter sw = new StreamWriter(writePath, false, System.Text.Encoding.Default))
* {
* sw.WriteLine(text);
* }
* using (StreamWriter sw = new StreamWriter(writePath, true, System.Text.Encoding.Default))
* {
* sw.WriteLine("Дозапись");
* sw.Write(4.5);
* }
* }
* catch (Exception e)
* {
* Console.WriteLine(e.Message);
* }

# BinaryWriter и BinaryReader

### Основные метода класса BinaryWriter

* **Close()**: закрывает поток и освобождает ресурсы
* **Flush()**: очищает буфер, дописывая из него оставшиеся данные в файл
* **Seek()**: устанавливает позицию в потоке
* **Write()**: записывает данные в поток

### Основные метода класса BinaryReader

* **Close()**: закрывает поток и освобождает ресурсы
* **ReadBoolean()**: считывает значение bool и перемещает указатель на один байт
* **ReadByte()**: считывает один байт и перемещает указатель на один байт
* **ReadChar()**: считывает значение char, то есть один символ, и перемещает указатель на столько байтов, сколько занимает символ в текущей кодировке
* **ReadDecimal()**: считывает значение decimal и перемещает указатель на 16 байт
* **ReadDouble()**: считывает значение double и перемещает указатель на 8 байт
* **ReadInt16()**: считывает значение short и перемещает указатель на 2 байта
* **ReadInt32()**: считывает значение int и перемещает указатель на 4 байта
* **ReadInt64()**: считывает значение long и перемещает указатель на 8 байт
* **ReadSingle()**: считывает значение float и перемещает указатель на 4 байта
* **ReadString()**: считывает значение string. Каждая строка предваряется значением длины строки, которое представляет 7-битное целое число
* struct State
* {
* public string name;
* public string capital;
* public int area;
* public double people;
* public State(string n, string c, int a, double p)
* {
* name = n;
* capital = c;
* people = p;
* area = a;
* }
* }
* class Program
* {
* static void Main(string[] args)
* {
* State[] states = new State[2];
* states[0] = new State("Германия", "Берлин",  357168,  80.8);
* states[1] = new State("Франция", "Париж", 640679, 64.7);
* string path= @"C:\SomeDir\states.dat";
* try
* {
* // создаем объект BinaryWriter
* using (BinaryWriter writer = new BinaryWriter(File.Open(path, FileMode.OpenOrCreate)))
* {
* // записываем в файл значение каждого поля структуры
* foreach (State s in states)
* {
* writer.Write(s.name);
* writer.Write(s.capital);
* writer.Write(s.area);
* writer.Write(s.people);
* }
* }
* // создаем объект BinaryReader
* using (BinaryReader reader = new BinaryReader(File.Open(path, FileMode.Open)))
* {
* // пока не достигнут конец файла
* // считываем каждое значение из файла
* while (reader.PeekChar() > -1)
* {
* string name = reader.ReadString();
* string capital = reader.ReadString();
* int area = reader.ReadInt32();
* double population = reader.ReadDouble();
* Console.WriteLine("Страна: {0}  столица: {1}  площадь {2} кв. км   численность населения: {3} млн. чел.",
* name, capital, area, population);
* }
* }
* }
* catch (Exception e)
* {
* Console.WriteLine(e.Message);
* }
* Console.ReadLine();
* }
* }

# Сериализация

### Формат сериализации

Хотя сериализация представляет собой преобразование объекта в некоторый набор байтов, но в действительности только бинарным форматом она не ограничивается. Итак, в .NET можно использовать следующие форматы:

* бинарный
* SOAP
* xml
* JSON

## BinaryFormatter

using System;

using System.IO;

using System.Runtime.Serialization.Formatters.Binary;

namespace Serialization

{

    [Serializable]

    class Person

    {

        public string Name { get; set; }

        public int Age { get; set; }

        public Person(string name, int age)

        {

            Name = name;

            Age = age;

        }

    }

    class Program

    {

        static void Main(string[] args)

        {

            // объект для сериализации

            Person person = new Person("Tom", 29);

            Console.WriteLine("Объект создан");

            // создаем объект BinaryFormatter

            BinaryFormatter formatter = new BinaryFormatter();

            // получаем поток, куда будем записывать сериализованный объект

            using (FileStream fs = new FileStream("people.dat", FileMode.OpenOrCreate))

            {

                formatter.Serialize(fs, person);

                Console.WriteLine("Объект сериализован");

            }

            // десериализация из файла people.dat

            using (FileStream fs = new FileStream("people.dat", FileMode.OpenOrCreate))

            {

                Person newPerson = (Person)formatter.Deserialize(fs);

                Console.WriteLine("Объект десериализован");

                Console.WriteLine("Имя: {0} --- Возраст: {1}", newPerson.Name, newPerson.Age);

            }

            Console.ReadLine();

        }

    }

}

Хотя мы взяли лишь один объект Person, но равным образом мы можем использовать и массив подобных объектов, список или иную коллекцию, к которой применяется атрибут Serializable. Посмотрим на примере массива

Person person1 = new Person("Tom", 29);

Person person2 = new Person("Bill", 25);

// массив для сериализации

Person[] people = new Person[] { person1, person2 };

BinaryFormatter formatter = new BinaryFormatter();

using (FileStream fs = new FileStream("people.dat", FileMode.OpenOrCreate))

{

    // сериализуем весь массив people

    formatter.Serialize(fs, people);

    Console.WriteLine("Объект сериализован");

}

// десериализация

using (FileStream fs = new FileStream("people.dat", FileMode.OpenOrCreate))

{

    Person[] deserilizePeople = (Person[])formatter.Deserialize(fs);

    foreach (Person p in deserilizePeople)

    {

        Console.WriteLine("Имя: {0} --- Возраст: {1}", p.Name, p.Age);

    }

}

## SoapFormatter

Протокол SOAP (Simple Object Access Protocol) представляет простой протокол для обмена данными между различными платформами. При такой сериализации данные упакуются в конверт SOAP, данные в котором имеют вид xml-подобного документа

Прежде чем использовать класс SoapFormatter, нам надо добавить в проект сборку *System.Runtime.Serialization.Formatters.Soap.dll*. После этого нам станет доступным функциональность SoapFormatter:

using System;

using System.IO;

using System.Runtime.Serialization.Formatters.Soap;

namespace Serialization

{

    [Serializable]

    class Person

    {

        public string Name { get; set; }

        public int Age { get; set; }

        public Person(string name, int age)

        {

            Name = name;

            Age = age;

        }

    }

    class Program

    {

        static void Main(string[] args)

        {

            Person person = new Person("Tom", 29);

            Person person2 = new Person("Bill", 25);

            Person[] people = new Person[] { person, person2 };

            // создаем объект SoapFormatter

            SoapFormatter formatter = new SoapFormatter();

            // получаем поток, куда будем записывать сериализованный объект

            using (FileStream fs = new FileStream("people.soap", FileMode.OpenOrCreate))

            {

                formatter.Serialize(fs, people);

                Console.WriteLine("Объект сериализован");

            }

            // десериализация

            using (FileStream fs = new FileStream("people.soap", FileMode.OpenOrCreate))

            {

                Person[] newPeople = (Person[])formatter.Deserialize(fs);

                Console.WriteLine("Объект десериализован");

                foreach (Person p in newPeople)

                {

                    Console.WriteLine("Имя: {0} --- Возраст: {1}", p.Name, p.Age);

                }

            }

            Console.ReadLine();

        }

    }

}

## XmlSerializer

ля сериализации объектов в файлы xml используется класс XmlSerializer. Он стоит несколько особняком от других ранее рассмотренных классов сериализаций, поэтому работа с ним будет немного отличаться.

Во-первых, XmlSerializer предполагает некоторые ограничения. Например, класс, подлежащий сериализации, должен иметь стандартный конструктор без параметров. Также сериализации подлежат только открытые члены. Если в классе есть поля или свойства с модификатором private, то при сериализации они будут игнорироваться

using System;

using System.IO;

using System.Xml.Serialization;

namespace Serialization

{

    // класс и его члены объявлены как public

    [Serializable]

    public class Person

    {

        public string Name { get; set; }

        public int Age { get; set; }

        // стандартный конструктор без параметров

        public Person()

        { }

        public Person(string name, int age)

        {

            Name = name;

            Age = age;

        }

    }

    class Program

    {

        static void Main(string[] args)

        {

            // объект для сериализации

            Person person = new Person("Tom", 29);

            Console.WriteLine("Объект создан");

            // передаем в конструктор тип класса

            XmlSerializer formatter = new XmlSerializer(typeof(Person));

            // получаем поток, куда будем записывать сериализованный объект

            using (FileStream fs = new FileStream("persons.xml", FileMode.OpenOrCreate))

            {

               formatter.Serialize(fs, person);

                Console.WriteLine("Объект сериализован");

            }

            // десериализация

            using (FileStream fs = new FileStream("persons.xml", FileMode.OpenOrCreate))

            {

                Person newPerson = (Person)formatter.Deserialize(fs);

                Console.WriteLine("Объект десериализован");

                Console.WriteLine("Имя: {0} --- Возраст: {1}", newPerson.Name, newPerson.Age);

            }

            Console.ReadLine();

        }

    }

}

Но это был простой объект. Однако с более сложными по составу объектами работать так же просто.

using System;

using System.IO;

using System.Xml.Serialization;

namespace Serialization

{

    [Serializable]

    public class Person

    {

        public string Name { get; set; }

        public int Age { get; set; }

        public Company Company { get; set; }

        public Person()

        { }

        public Person(string name, int age, Company comp)

        {

            Name = name;

            Age = age;

            Company = comp;

        }

    }

    [Serializable]

    public class Company

    {

        public string Name { get; set; }

        // стандартный конструктор без параметров

        public Company() { }

        public Company(string name)

        {

            Name = name;

        }

    }

    class Program

    {

        static void Main(string[] args)

        {

            Person person1 = new Person("Tom", 29, new Company("Microsoft"));

            Person person2 = new Person("Bill", 25, new Company("Apple"));

            Person[] people = new Person[] { person1, person2 };

            XmlSerializer formatter = new XmlSerializer(typeof(Person[]));

            using (FileStream fs = new FileStream("people.xml", FileMode.OpenOrCreate))

            {

               formatter.Serialize(fs, people);

            }

            using (FileStream fs = new FileStream("people.xml", FileMode.OpenOrCreate))

            {

                Person[] newpeople = (Person[])formatter.Deserialize(fs);

                foreach (Person p in newpeople)

                {

                    Console.WriteLine("Имя: {0} --- Возраст: {1} --- Компания: {2}", p.Name, p.Age, p.Company.Name);

                }

            }

            Console.ReadLine();

        }

    }

}

## DataContractJsonSerializer

Для сериализации объектов в формат JSON в пространстве System.Runtime.Serialization.Json определен класс DataContractJsonSerializer. Чтобы задействовать этот класс, в проект надо добавить сборку *System.Runtime.Serialization.dll*. Для записи объектов в json-файл в этом классе имеется метод WriteObject(), а для чтения ранее сериализованных объектов - метод ReadObject()

using System;

using System.IO;

using System.Runtime.Serialization.Json;

using System.Runtime.Serialization;

namespace Serialization

{

    [DataContract]

    public class Person

    {

        [DataMember]

        public string Name { get; set; }

        [DataMember]

        public int Age { get; set; }

        public Person(string name, int year)

        {

            Name = name;

            Age = year;

        }

    }

    class Program

    {

        static void Main(string[] args)

        {

            // объект для сериализации

            Person person1 = new Person("Tom", 29);

            Person person2 = new Person("Bill", 25);

            Person[] people = new Person[] { person1, person2 };

            DataContractJsonSerializer jsonFormatter = new DataContractJsonSerializer(typeof(Person[]));

            using (FileStream fs = new FileStream("people.json", FileMode.OpenOrCreate))

            {

                jsonFormatter.WriteObject(fs, people);

            }

            using (FileStream fs = new FileStream("people.json", FileMode.OpenOrCreate))

            {

                Person[] newpeople = (Person[])jsonFormatter.ReadObject(fs);

                foreach (Person p in newpeople)

                {

                    Console.WriteLine("Имя: {0} --- Возраст: {1}", p.Name, p.Age);

                }

            }

            Console.ReadLine();

        }

    }

}

# Работа со строками

string s1 = "hello";

string s2 = null;

string s3 = new String('a', 6); // результатом будет строка "aaaaaa"

string s4 = new String(new char[]{'w', 'o', 'r', 'l', 'd'});

### Основные методы строк

Основная функциональность класса String раскрывается через его методы, среди которых можно выделить следующие:

* **Compare**: сравнивает две строки с учетом текущей культуры (локали) пользователя
* **CompareOrdinal**: сравнивает две строки без учета локали
* **Contains**: определяет, содержится ли подстрока в строке
* **Concat**: соединяет строки
* **CopyTo**: копирует часть строки или всю строку в другую строку
* **EndsWith**: определяет, совпадает ли конец строки с подстрокой
* **Format**: форматирует строку
* **IndexOf**: находит индекс первого вхождения символа или подстроки в строке
* **Insert**: вставляет в строку подстроку
* **Join**: соединяет элементы массива строк
* **LastIndexOf**: находит индекс последнего вхождения символа или подстроки в строке
* **Replace**: замещает в строке символ или подстроку другим символом или подстрокой
* **Split**: разделяет одну строку на массив строк
* **Substring**: извлекает из строки подстроку, начиная с указанной позиции
* **ToLower**: переводит все символы строки в нижний регистр
* **ToUpper**: переводит все символы строки в верхний регистр
* **Trim**: удаляет начальные и конечные пробелы из строки

string[] words = text.Split(new char[] { ' ' }, StringSplitOptions.RemoveEmptyEntries);

## Форматирование и интерполяция строк

C / c Задает формат денежной единицы, указывает количество десятичных разрядов после запятой

D / d Целочисленный формат, указывает минимальное количество цифр

E / e Экспоненциальное представление числа, указывает количество десятичных разрядов после запятой

F / f Формат дробных чисел с фиксированной точкой, указывает количество десятичных разрядов после запятой

G / g Задает более короткий из двух форматов: F или E

N / n Также задает формат дробных чисел с фиксированной точкой, определяет количество разрядов после запятой

P / p Задает отображения знака процентов рядом с число, указывает количество десятичных разрядов после запятой

X / x Шестнадцатеричный формат числа

double number = 23.7;

string result = String.Format("{0:C}", number);

Console.WriteLine(result); // $ 23.7

string result2 = String.Format("{0:C2}", number);

Console.WriteLine(result2); // $ 23.70

int number = 23;

string result = String.Format("{0:d}", number);

Console.WriteLine(result); // 23

string result2 = String.Format("{0:d4}", number);

Console.WriteLine(result2); // 0023

int number = 23;

string result = String.Format("{0:f}", number);

Console.WriteLine(result); // 23,00

double number2 = 45.08;

string result2 = String.Format("{0:f4}", number2);

Console.WriteLine(result2); // 45,0800

double number3 = 25.07;

string result3 = String.Format("{0:f1}", number3);

Console.WriteLine(result2); // 25,1

decimal number = 0.15345m;

Console.WriteLine("{0:P1}", number);// 15.3%

эlong number = 19876543210;

string result = String.Format("{0:+# (###) ###-##-##}", number);

Console.WriteLine(result); // +1 (987) 654-32-10

### интерполяция строк

Начиная с версии языка C# 6.0 (Visual Studio 2015) была добавлена такая функциональность, как интерполяция строк. Эта функциональность призвана заменить форматирование строк. Так, перепишем пример с выводом значений свойств объекта Person:

Person person = new Person { Name = "Tom", Age = 23 };

Console.WriteLine($"Имя: {person.Name}  Возраст: {person.Age}");

int x = 8;

int y = 7;

string result = $"{x} + {y} = {x + y}";

Console.WriteLine(result); // 8 + 7 = 15

В следующем примере проверяем, не равен ли person значению null. Если не равен, то выводим его имя, иначе выводим какое-нибудь имя по умолчанию:

|  |  |
| --- | --- |
| 1  2  3 | person = null;  string output = $"{person?.Name??"Имя по умолчанию"}";  Console.WriteLine(output); |

Уже внутри строки можно применять форматирование. В этом случае мы можем применять все те же описатели, что и в методе Format. Например, выведем номер телефона в формате +x xxx-xxx-xx-xx:

|  |  |
| --- | --- |
| 1  2 | long number = 19876543210;  Console.WriteLine($"{number:+# ### ### ## ##}"); // +1 987 654 32 10 |

Добавляем пространство до и после форматируемого вывода:

|  |  |
| --- | --- |
| 1  2 | Console.WriteLine($"Имя: {person.Name, -5} Возраст: {person.Age}"); // пробелы после  Console.WriteLine($"Имя: {person.Name, 5} Возраст: {person.Age}"); // пробелы до |

## StringBuilder

При создании строки StringBuilder выделяет памяти больше, чем необходимо этой строке:

StringBuilder sb = new StringBuilder("Привет мир");

Console.WriteLine("Длина строки: {0}", sb.Length);

Console.WriteLine("Емкость строки: {0}", sb.Capacity);

Кроме метода Append класс StringBuilder предлагает еще ряд методов для операций над строками:

* **Insert**: вставляет подстроку в объект StringBuilder, начиная с определенного индекса
* **Remove**: удаляет определенное количество символов, начиная с определенного индекса
* **Replace**: заменяет все вхождения определенного символа или подстроки на другой символ или подстроку
* **AppendFormat**: доабвляет подстроку в конец объекта StringBuilder
* StringBuilder sb = new StringBuilder("Привет мир");
* sb.Append("!");
* sb.Insert(7, "компьютерный ");
* Console.WriteLine(sb);
* // заменяем слово
* sb.Replace("мир", "world");
* Console.WriteLine(sb);
* // удаляем 13 символов, начиная с 7-го
* sb.Remove(7, 13);
* Console.WriteLine(sb);
* // получаем строку из объекта StringBuilder
* string s = sb.ToString();
* Console.WriteLine(s);

Microsoft рекомендует использовать класс String в следующих случаях:

* При небольшом количестве операций и изменений над строками
* При выполнении фиксированного количества операций объединения. В этом случае компилятор может объединить все операции объединения в одну
* Когда надо выполнять масштабные операции поиска при построении строки, например IndexOf или StartsWith. Класс StringBuilder не имеет подобных методов.

Класс StringBuilder рекомендуется использовать в следующих случаях:

* При неизвестном количестве операций и изменений над строками во время выполнения программы
* Когда предполагается, что приложению придется сделать множество подобных операций

# Регулярные выражения

string s = "Бык тупогуб, тупогубенький бычок, у быка губа бела была тупа";

Regex regex = new Regex(@"туп(\w\*)");

MatchCollection matches = regex.Matches(s);

if (matches.Count > 0)

{

    foreach (Match match in matches)

        Console.WriteLine(match.Value);

}

else

{

    Console.WriteLine("Совпадений не найдено");

}

### Параметр RegexOptions

Класс Regex имеет ряд конструкторов, позволяющих выполнить начальную инициализацию объекта. Две версии конструкторов в качестве одного из параметров принимают перечисление RegexOptions. Некоторые из значений, принимаемых данным перечислением:

* **Compiled**: при установке этого значения регулярное выражение компилируется в сборку, что обеспечивает более быстрое выполнение
* **CultureInvariant**: при установке этого значения будут игнорироваться региональные различия
* **IgnoreCase**: при установке этого значения будет игнорироваться регистр
* **IgnorePatternWhitespace**: удаляет из строки пробелы и разрешает комментарии, начинающиеся со знака #
* **Multiline**: указывает, что текст надо рассматривать в многострочном режиме. При таком режиме символы "^" и "$" совпадают, соответственно, с началом и концом любой строки, а не с началом и концом всего текста
* **RightToLeft**: приписывает читать строку справа налево
* **Singleline**: устанавливает однострочный режим, а весь текст рассматривается как одна строка

Regex regex = new Regex(@"туп(\w\*)", RegexOptions.IgnoreCase);

Regex regex = new Regex(@"туп(\w\*)", RegexOptions.Compiled | RegexOptions.IgnoreCase);

### Синтаксис регулярных выражений

https://msdn.microsoft.com/ru-ru/library/az24scfc(v=vs.110).aspx

Рассмотрим вкратце некоторые элементы синтаксиса регулярных выражений:

* **^**: соответствие должно начинаться в начале строки (например, выражение @"^пр\w\*"соответствует строке "привет мир")
* **$**: конец строки (например, выражение @"\w\*ир$" соответствует строке "привет мир", так как часть "ир" находится в самом конце)
* **.**: знак точки определяет любой одиночный символ (например, выражение "м.р"соответствует слову "мир" или "мор")
* **\***: предыдущий символ повторяется 0 и более раз
* **+**: предыдущий символ повторяется 1 и более раз
* **?**: предыдущий символ повторяется 0 или 1 раз
* **\s**: соответствует любому пробельному символу
* **\S**: соответствует любому символу, не являющемуся пробелом
* **\w**: соответствует любому алфавитно-цифровому символу
* **\W**: соответствует любому не алфавитно-цифровому символу
* **\d**: соответствует любой десятичной цифре
* **\D**: соответствует любому символу, не являющемуся десятичной цифрой

string s = "456-435-2318";

Regex regex = new Regex(@"\d{3}-\d{3}-\d{4}");

Если мы точно знаем, сколько определенных символов должно быть, то мы можем явным образом указать их количество в фигурных скобках: \d{3} - то есть в данном случае три цифры.

string s = "456-435-2318";

Regex regex = new Regex("[0-9]{3}-[0-9]{3}-[0-9]{4}");

В квадратных скобках задается диапазон символов, которые должны в данном месте встречаться. В итоге данный и предыдущий шаблоны телефонного номера будут эквивалентны.

### Замена и метод Replace

Класс Regex имеет метод Replace, который позволяет заменить строку, соответствующую регулярному выражению, другой строкой:

|  |  |
| --- | --- |
| 1  2  3  4  5 | string s = "Мама  мыла  раму. ";  string pattern = @"\s+";  string target = " ";  Regex regex = new Regex(pattern);  string result = regex.Replace(s, target); |

# Сборка мусора, управление памятью и указатели

Рассмотрим некоторые методы и свойства класса System.GC:

* Метод **AddMemoryPressure** информирует среду CLR о выделении большого объема неуправляемой памяти, которую надо учесть при планировании сборки мусора. В связке с этим методом используется метод **RemoveMemoryPressure**, который указывает CLR, что ранее выделенная память освобождена, и ее не надо учитывать при сборке мусора.
* Метод **Collect** приводит в действие механизм сборки мусора. Перегруженные версии метода позволяют указать поколение объектов, вплоть до которого надо произвести сборку мусора
* Метод **GetGeneration(Object)** позволяет определить номер поколения, к которому относится переданый в качестве параметра объект
* Метод **GetTotalMemory** возвращает объем памяти в байтах, которое занято в управляемой куче
* Метод **WaitForPendingFinalizers** приостанавливает работу текущего потока до освобождения всех объектов, для которых производится сборка мусора

Однако на деле при очистке сборщик мусора вызывает не деструктор, а метод **Finalize** класса Person. Все потому, что компилятор C# компилирует деструктор в конструкцию, которая эквивалентна следующей

protected override void Finalize()

{

    try

    {

        // здесь идут инструкции деструктора

    }

    finally

    {

        base.Finalize();

    }

}

class Program

{

    static void Main(string[] args)

    {

        Test();

        Console.ReadLine();

    }

    private static void Test()

    {

        Person p = new Person();

    }

}

Обратите внимание, что даже после завершения метода Test и соответственно удаления из стека ссылки на объект Person в куче, может не последовать немедленного вызова деструктора. Лишь при завершении всей программы гарантировано произойдет очистка памяти и вызов деструктора.

На уровне памяти это выглядит так: сборщик мусора при размещении объекта в куче определяет, поддерживает ли данный объект метод Finalize. И если объект имеет метод Finalize, то указатель на него сохраняется в специальной таблице, которая называется очередь финализации. Когда наступает момент сборки мусора, сборщик видит, что данный объект должен быть уничтожен, и если он имеет метод Finalize, то он копируется в еще одну таблицу и окончательно уничтожается лишь при следующем проходе сборщика мусора.

И здесь мы можем столкнуться со следующей проблемой: а что если нам немедленно надо вызвать деструктор и освободить все связанные с объектом неуправляемые ресурсы? В этом случае мы можем использовать второй подход - реализацию интерфейса IDisposable.

### Интерфейс IDisposable

Интерфейс IDisposable объявляет один единственный метод Dispose, в котором при реализации интерфейса в классе должно происходить освобождение неуправляемых ресурсов.

class Program

{

    static void Main(string[] args)

    {

        Test();

        Console.ReadLine();

    }

    private static void Test()

    {

        Person p=null;

        try

        {

            p = new Person();

        }

        finally

        {

            if (p != null)

            {

                p.Dispose();

            }

        }

    }

}

public class Person : IDisposable

{

    public void Dispose()

    {

        Console.Beep();

    }

}

Но конструкцию try...finally предпочтительнее использовать при вызове метода Dispose, так как она гарантирует, что даже в случае возникновения исключения произойдет освобождение ресурсов в методе Dispose.

## Указатели

Чтобы использовать небезопасный код в C#, надо первым делом указать проекту, что он будет работать с небезопасным кодом. Для этого надо установить в настройках проекта соответствующий флаг - в меню Project (Проект) найти Свойства проекта. Затем в меню Build установить флажок **Allow unsafe code (Разрешить небезопасный код).**

### Операции \* и &

Ключевой при работе с указателями является операция \*, которую еще называют операцией разыменовывания. Операция разыменовывания позволяет получить или установить значение по адресу, на который указывает указатель. Для получения адреса переменной применяется операция &

static void Main(string[] args)

{

    unsafe

    {

        int\* x; // определение указателя

        int y = 10; // определяем переменную

        x = &y; // указатель x теперь указывает на адрес переменной y

        Console.WriteLine(\*x); // 10

        y = y + 20;

         Console.WriteLine(\*x);// 30

        \*x = 50;

        Console.WriteLine(y); // переменная y=50

    }

    Console.ReadLine();

}

### Получение адреса

Используя преобразование указателя к целочисленному типу, можно получить адрес памяти, на который указывает указатель

int\* x; // определение указателя

int y = 10; // определяем переменную

x = &y; // указатель x теперь указывает на адрес переменной y

// получим адрес переменной y

uint addr = (uint)x;

Console.WriteLine("Адрес переменной y: {0}", addr);

### Операции с указателями

Кроме операции разыменовывания к указателям применимы еще и некоторые арифметические операции( +, ++, -, --, +=, -=) и преобразования. Например, мы можем преобразовать число в указатель

int\* x; // определение указателя

int y = 10; // определяем переменную

x = &y; // указатель x теперь указывает на адрес переменной y

// получим адрес переменной y

uint addr = (uint)x;

Console.WriteLine("Адрес переменной y: {0}", addr);

byte\* bytePointer = (byte\*)(addr+4); // получить указатель на следующий байт после addr

Console.WriteLine("Значение byte по адресу {0}: {1}", addr+4, \*bytePointer);

// обратная операция

uint oldAddr = (uint)bytePointer - 4; // вычитаем четыре байта, так как bytePointer - указатель на байт

int\* intPointer = (int\*)oldAddr;

Console.WriteLine("Значение int по адресу {0}: {1}", oldAddr, \*intPointer);

// преобразование в тип double

double\* doublePointer = (double\*)(addr + 4);

Console.WriteLine("Значение double по адресу {0}: {1}", addr + 4, \*doublePointer);

### Указатели на массивы и stackalloc

С помощью ключевого слова **stackalloc** можно выделить память под массив в стеке. Смысл выделения памяти в стеке в повышении быстродействия кода. Посмотрим на примере вычисления факториала

unsafe

{

    const int size = 7;

    int\* factorial = stackalloc int[size]; // выделяем память в стеке под семь объектов int

    int\* p = factorial;

    \*(p++)= 1; // присваиваем первой ячейке значение 1 и

    // увеличиваем указатель на 1

    for (int i = 2; i <= size; i++, p++)

    {

        // считаем факториал числа

        \*p = p[-1] \*i;

    }

    for (int i = 1; i <= size; ++i)

    {

        Console.WriteLine(factorial[i-1]);

    }

}

### Оператор fixed и закрепление указателей

Ранее мы посмотрели, как создавать указатели на типы значений, например, int или структуры. Однако кроме структур в C# есть еще и классы, которые в отличие от типов значений, помещают все связанные значения в куче. И в работу данных классов может в любой момент вмешаться сборщик мусора, периодически очищающий кучу. Чтобы фиксировать на все время работы указатели на объекты классов используется оператор fixed.

Допустим, у нас есть класс Person:

|  |  |
| --- | --- |
| 1  2  3  4  5 | public class Person  {      public int age;      public int height;  } |

Зафиксируем указатель с помощью оператора fixed:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15 | unsafe  {      Person person = new Person();      person.age = 28;      person.height = 178;      // блок фиксации указателя      fixed(int\* p = &person.age)      {          if (\*p < 30)          {              \*p = 30;          }      }      Console.WriteLine(person.age); // 30  } |

# Dynamic Language Runtime

Ключевым моментом использования DLR в C# является применение типов **dynamic**. Это ключевое слово позволяет опустить проверку типов во время компиляции. Кроме того, объекты, объявленные как dynamic, могут в течение работы программы менять свой тип

class Program

{

    static void Main(string[] args)

    {

        dynamic x = 3; // здесь x - целочисленное int

        Console.WriteLine(x);

        x = "Привет мир"; // x - строка

        Console.WriteLine(x);

        x = new Person() { Name = "Tom", Age = 23 }; // x - объект Person

        Console.WriteLine(x);

        Console.ReadLine();

    }

}

class Person

{

    public string Name {get;set;}

    public int Age { get; set; }

Несмотря на то, что переменная x меняет тип своего значения несколько раз, данный код будет нормально работать. В этом использование типов dynamic отличается от применения ключевого слова **var**. Для переменной, объявленной с помощью ключевого слова var, тип выводится во время компиляции и затем во время выполнения больше не меняется.

Также можно найти общее между использованием dynamic и типом object. Если в предыдущем примере мы заменим dynamic на object:object x = 3;, то результат будет тот же. Однако и тут есть различия. Например:

|  |  |
| --- | --- |
| 1  2  3  4 | object obj = 24;  dynamic dyn = 24;  obj += 4; // так нельзя  dyn += 4; // а так можно |

На строке obj += 4; мы увидим ошибку, так как операция += не может быть применена к типам object и int. С переменной, объявленной как dynamic, это пройдет, так как ее тип будет известен только во время выполнения.

Еще одна отличительная особенность использования dynamic состоит в том, что это ключевое слово применяется не только к переменным, но и к свойствам и методам. Например:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27 | class Person  {      public string Name {get;set;}      public dynamic Age { get; set; }        // выводим зарплату в зависимости от переданного формата      public dynamic getSalary(dynamic value, string format)      {          if (format=="string")          {              return value + " рублей";          }          else if (format == "int")          {              return value;          }          else          {              return 0.0;          }      } |

Также есть метод getSalary, возвращающий значение dynamic. Например, в зависимости от параметра мы можем вернуть или строковое представление суммы дохода или численное. Также метод принимает dynamic в качестве параметра. Таким образом, мы можем передать в качестве значения дохода как целое, так и дробное число

# DynamicObject и ExpandoObject

Интересные возможности при разработке в C# и .NET с использованием DLR предоставляет пространство имен **System.Dynamic** и в частности класс **ExpandoObject**. Он позволяет создавать динамические объекты, наподобие тех, что используются в javascript.

dynamic viewbag = new System.Dynamic.ExpandoObject();

viewbag.Name = "Tom";

viewbag.Age = 46;

viewbag.Languages = new List<string> {"english", "german", "french" };

Console.WriteLine("{0} - {1}", viewbag.Name, viewbag.Age);

foreach (var lang in viewbag.Languages)

    Console.WriteLine(lang);

// объявляем метод

viewbag.IncrementAge = (Action<int>)(x => viewbag.Age += x);

viewbag.IncrementAge(6); // увеличиваем возраст на 6 лет

Console.WriteLine("{0} - {1}", viewbag.Name, viewbag.Age);

### DynamicObject

На ExpandoObject по своему действию похож другой класс - DynamicObject. Он также позволяет задавать динамические объекты. Только в данном случае нам надо создать свой класс, унаследовав его от DynamicObject и реализовав его методы:

* TryBinaryOperation(): выполняет бинарную операцию между двумя объектами. Эквивалентно стандартным бинарным операциям, например, сложению x + y)
* TryConvert(): выполняет преобразование к определенному типу. Эквивалентно базовому преобразованию в C#, например, (SomeType) obj
* TryCreateInstance(): создает экземпляр объекта
* TryDeleteIndex(): удаляет индексатор
* TryDeleteMember(): удаляет свойство или метод
* TryGetIndex(): получает элемент по индексу через индексатор. В C# может быть эквивалентно следующему выражению int x = collection[i]
* TryGetMember(): получаем значение свойства. Эквивалентно обращению к свойству, например, string n = person.Name
* TryInvoke(): вызов объекта в качестве делегата
* TryInvokeMember(): вызов метода
* TrySetIndex(): устанавливает элемент по индексу через индексатор. В C# может быть эквивалентно следующему выражению collection[i] = x;
* TrySetMember(): устанавливает свойство. Эквивалентно присвоению свойству значения person.Name = "Tom"
* TryUnaryOperation(): выполняет унарную операцию подобно унарным операциям в C#: x++

# Использование IronPython в .NET

ассмотрим на примере применение IronPython. Но для начала необходимо добавить в проект несколько пакетов через пакетный менеджер NuGet. Для того нажмем в окне проекта на узел References правой кнопкой мыши и выберем в появившемся списке пункт **Manage NuGet Packages**

И перед нами откроется окно пакетного менеджера. Чтобы найти нужный пакет, введем в поле поиска "DLR", и менеджер отобразит ряд результатов, из которых первый - пакет **DynamicLanguageRuntime** необходимо установить

После этого в проект в узел References добавляется библиотека **Microsoft.Scripting**.

Теперь также нам надо добавить необходимые пакеты для IronPython. Для этого введем в поле поиска "IronPython" и после этого установим одноименный пакет

using System;

using IronPython.Hosting;

using Microsoft.Scripting.Hosting;

namespace IronPythonApp

{

    class Program

    {

        static void Main(string[] args)

        {

            ScriptEngine engine = Python.CreateEngine();

            engine.Execute("print 'hello, world'");

        }

    }

}

Мы также могли бы определить файл *hello.py*, то есть обычный текстовый файл с кодом на языке Python, со следующим содержимым:

|  |  |
| --- | --- |
| 1 | print 'hello, world' |

И запустить его в программе:

|  |  |
| --- | --- |
| 1  2  3 | ScriptEngine engine = Python.CreateEngine();  engine.ExecuteFile("[D://hello.py](file:///D:\hello.py)");  Console.Read(); |

### ScriptScope

Объект ScriptScope позволяет взаимодействовать со скриптом, получая или устанавливая его переменные, получая ссылки на функции. Например, напишем простейший скрипт *hello2.py*, который использует переменные:

|  |  |
| --- | --- |
| 1  2  3 | x = 10  z = x + y  print z |

Теперь напишем программу, которая будет взаимодействовать со скриптом:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | int yNumber = 22;  ScriptEngine engine = Python.CreateEngine();  ScriptScope scope = engine.CreateScope();  scope.SetVariable("y", yNumber);  engine.ExecuteFile("[D://hello2.py](file:///D:\hello2.py)", scope);  dynamic xNumber = scope.GetVariable("x");  dynamic zNumber = scope.GetVariable("z");  Console.WriteLine("Сумма {0} и {1} равна: {2}", xNumber, yNumber, zNumber); |

### Вызов функций из IronPython

Определим скрипт *factorial.py*, который содержит функцию, вычисляющую факториал числа:

|  |  |
| --- | --- |
| 1  2  3  4  5 | def factorial(number):      result = 1      for i in xrange(2, number + 1):          result \*= i      return result |

Теперь обратимся к этой функции в коде C#:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26 | using System;  using IronPython.Hosting;  using Microsoft.Scripting.Hosting;    namespace IronPythonApp  {      class Program      {          static void Main(string[] args)          {              Console.WriteLine("Введите число:");              int x = Int32.Parse(Console.ReadLine());                ScriptEngine engine = Python.CreateEngine();              ScriptScope scope = engine.CreateScope();                engine.ExecuteFile("[D://factorial.py](file:///D:\factorial.py)", scope);              dynamic function = scope.GetVariable("factorial");              // вызываем функцию и получаем результат              dynamic result = function(x);              Console.WriteLine(result);                Console.Read();          }      }  } |

Получить объект функции можно также, как и переменную: scope.GetVariable("factorial");. Затем с этим объектом работаем также, как и с любым другим методом

# Сборки .NET

Сборки имеют следующие составляющие:

* Манифест, который содержит метаданные сборки
* Метаданные типов. Используя эти метаданные, сборка определяет местоположение типов в файле приложения, а также места размещения их в памяти
* Собственно код приложения на языке MSIL, в который компилируется код C#
* Ресурсы

Все эти компоненты могут находиться в одном файле, и тогда сборка представляет один единственный файл в формате exe или dll.

### Манифест сборки

Ключевым компонентом сборки является ее **манифест**. Если у сборки отсутствует манифест, то заключенный в ней код MSIL выполняться не будет. Манифест может находиться в одном файле с исполняемым кодом сборки, а может размещаться и в отдельном файле.

Манифест хранит следующие данные:

* **Имя сборки**
* **Номер версии**: основной и дополнительный номера. Используется для управления версиями
* **Язык и региональные параметры**: информация о языке и региональных параметрах, которые поддерживает сборка
* **Информация о строгом имени**: открытый ключ издателя
* **Список всех файлов сборки**: хэш и имя каждого из входящих в сборку файлов
* **Список ссылок на другие сборки**, которые использует текущая сборка
* **Список ссылок на типы**, используемые сборкой

# Разделяемые сборки. Добавление сборки в GAC

По способу взаимодействия с другими сборками и приложениями сборки можно разделить на две категории: закрытые и разделяемые.

Закрытые сборки это обычные сборки приложения, которые мы создаем в Visual Studio. Например, при создании библиотеки классов dll создается закрытая сборка. Впоследствии эту закрытую сборку мы можем использовать, подключив ее к другому проекту. А чтобы подключить к другому проекту, эту сборку можно просто положить рядом с исполняемым файлом и добавить в проект ссылку на нее через Add Reference. И на одной машине может быть десяток приложений, которые используют разные копии одной и той же сборки.

Если мы захотим удалить приложение, мы можем также удалить и используемую им закрытую сборку, и это не скажется на работе других приложений на локальной машине.

Иначе обстоит дело с разделяемыми сборками. По умолчанию при создании проекта visual Studio уже добавляет в проект ссылки на ряд разделяемых сборок. Открыв узел References (Ссылки). Например, Microsoft.CSharp.dll, System.dll, System.Core.dll - это все разделяемые сборки.

Разделяемые сборки находятся в глобальном кэше сборок (Global Assembly Cache). Местоположение кэша сборок отличается в зависимости от версии .NET, установленной на локальной машине. До .NET 4.0 глобальный кэш находился в каталоге *C:\Windows\assembly*. Начиная же с версии .NET 4.0 кэш сборок размещается по пути *C:\Windows\Microsoft.NET\assembly\GAC\_MSIL*

### Строгое имя сборки

Чтобы поместить сборку в GAC (глобальный кэш), эта сборка должна обладать строгим именем. В состав строгого имени входят следующие компоненты:

* Имя сборки без расширения
* Номер версии. Благодаря разграничению по версии можно использовать разные версии одной и ой же сборки
* Открытый ключ
* Необязательное значение для культуры (при локализации сборки)
* Цифровая подпись, которая создается с помощью хэш-значения содержимого сборки и значения секретного ключа. Секретный ключ представляет собой файл с расширением *\*.snk*.

Благодаря строгому имени гарантируется уникальность сборки в глобальном кэше.

Чтобы создать строгое имя, можно воспользоваться инструментарием, который имеется в Visual Studio. Допустим, мы создали проект по типу Class Library (Библиотека классов). И теперь мы хотим подписать сборку, которая будет компилироваться, строгим именем. Для этого нажмем в окне Solution Explorer (Обозреватель решений) на имя проекта правой кнопкой мыши и в появившемся меню выберем пункт Properties (Свойства). На вкладке свойств выберем пункт Signing:

Отметим флажок *Sign the assembly (Подписать сборку)*, как показано на рисунке. И чтобы создать новый секретный ключ, выберем в выпадающем списке пункт **New**. После этого откроется окно настроек секретного ключа:

Дадим новому ключу какое-нибудь имя и нажмем ОК. После этого в структуре проекта можно будет увидеть файл ключа

После задания сборке строгого имени ее можно добавлять в GAC. Для этого воспользуемся утилитой, которая идет в комплекте с .NET Framework, под названием *gacutil.exe*.

Откроем командную строку под администратором. Во-первых, найдем расположение утилиты gacutil.exe на локальной машине. У меня, например, она расположена в каталоге *C:\Program Files\Microsoft SDKs\Windows\v8.0A\bin\NETFX 4.0 Tools*. И вначале перейдем в этот каталог:

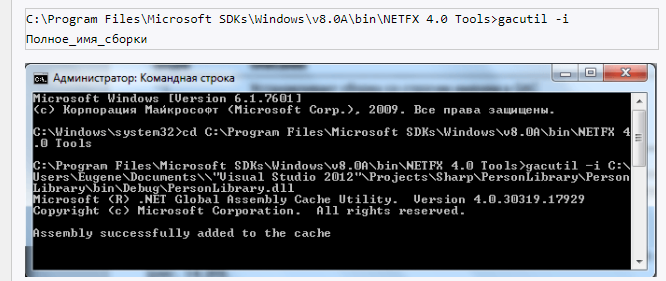
C:\Windows\system32>cd C:\Program Files\Microsoft SDKs\Windows\v8.0A\bin\NETFX 4.0 Tools

Теперь воспользуемся одной из команд данной утилиты. Наиболее используемые команды:

* *-i имя\_сборки* - установка сборки в GAC
* *-l* - вывод всего списка сборок в GAC
* *-u имя\_сборки* - удаление сборки из GAC

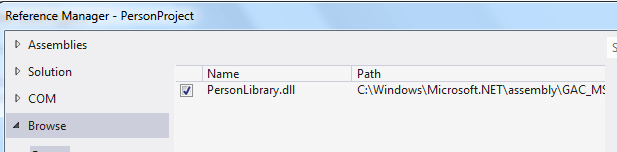
Теперь вводим в командной строке команду на добавление:

C:\Program Files\Microsoft SDKs\Windows\v8.0A\bin\NETFX 4.0 Tools>gacutil -i Полное\_имя\_сборки



И после добавления в папке *C:\Windows\Microsoft.NET\assembly\GAC\_MSIL* мы сможем найти добавленную сборку - для нее будет создан отдельный каталог, как и для остальных сборок, который будет носить краткое имя сборки.

Теперь мы можем использовать нашу сборку из GAC. Для этого создадим какой-нибудь проект и в окне Solution Exlplorer (Обозреватель решений) нажмем на узел Referenses (Ссылки). В появившемся меню выберем Add Reference... (Добавить ссылку):



# Многопоточность

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

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

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

Некоторые методы класса Thread:

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

### Статус потока

Статусы потока содержатся в перечислении **ThreadState**:

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

### Приоритеты потоков

Приоритеты поток располагаются в перечислении **ThreadPriority**:

* **Lowest**
* **BelowNormal**
* **Normal**
* **AboveNormal**
* **Highest**

## Создание потоков. Делегат ThreadStart

  // создаем новый поток

        Thread myThread = new Thread(new ThreadStart(Count));

        myThread.Start(); // запускаем поток

## Потоки с параметрами и ParameterizedThreadStart

static void Main(string[] args)

    {

        int number = 4;

        // создаем новый поток

        Thread myThread = new Thread(new ParameterizedThreadStart(Count));

        myThread.Start(number);

        for (int i = 1; i < 9; i++)

        {

            Console.WriteLine("Главный поток:");

            Console.WriteLine(i \* i);

            Thread.Sleep(300);

        }

        Console.ReadLine();

    }

    public static void Count(object x)

    {

        for (int i = 1; i < 9; i++)

        {

            int n = (int)x;

            Console.WriteLine("Второй поток:");

            Console.WriteLine(i\*n);

            Thread.Sleep(400);

        }

    }

Но тут опять же есть одно ограничение: метод Thread.Start не является типобезопасным, то есть мы можем передать в него любой тип, и потом нам придется приводить переданный объект к нужному нам типу. Для решения данной проблемы рекомендуется объявлять все используемые методы и переменные в специальном классе, а в основной программе запускать поток через ThreadStart.

class Program

{

    static void Main(string[] args)

    {

        Counter counter = new Counter(5, 4);

        Thread myThread = new Thread(new ThreadStart(counter.Count));

        myThread.Start();

        //........................

    }

}

public class Counter

{

    private int x;

    private int y;

    public Counter(int \_x, int \_y)

    {

        this.x = \_x;

        this.y = \_y;

    }

    public void Count()

    {

        for (int i = 1; i < 9; i++)

        {

            Console.WriteLine("Второй поток:");

            Console.WriteLine(i \* x \* y);

            Thread.Sleep(400);

        }

    }

}

## Синхронизация потоков

Решение проблемы состоит в том, чтобы синхронизировать потоки и ограничить доступ к разделяемым ресурсам на время их использования каким-нибудь потоком. Для этого используется ключевое слово **lock**. Оператор lock определяет блок кода, внутри которого весь код блокируется и становится недоступным для других потоков до завершения работы текущего потока. И мы можем переделать предыдущий пример следующим образом

class Program

{

    static int x=0;

    static object locker = new object();

    static void Main(string[] args)

    {

        for (int i = 0; i < 5; i++)

        {

            Thread myThread = new Thread(Count);

            myThread.Name = "Поток " + i.ToString();

            myThread.Start();

        }

        Console.ReadLine();

    }

    public static void Count()

    {

        lock (locker)

        {

            x = 1;

            for (int i = 1; i < 9; i++)

            {

                Console.WriteLine("{0}: {1}", Thread.CurrentThread.Name, x);

                x++;

                Thread.Sleep(100);

            }

        }

    }

}

## Мониторы

class Program

{

    static int x=0;

    static object locker = new object();

    static void Main(string[] args)

    {

        for (int i = 0; i < 5; i++)

        {

            Thread myThread = new Thread(Count);

            myThread.Name = "Поток " + i.ToString();

            myThread.Start();

        }

        Console.ReadLine();

    }

    public static void Count()

    {

        Monitor.Enter(locker);

        try

        {

            x = 1;

            for (int i = 1; i < 9; i++)

            {

                Console.WriteLine("{0}: {1}", Thread.CurrentThread.Name, x);

                x++;

                Thread.Sleep(100);

            }

        }

        finally

        {

            Monitor.Exit(locker);

        }

    }

}

Debug->Windows->Threads  
тут можно видеть все потоки процесса, переключаться и видеть стэк каждого потока

## AutoResetEvent

Класс AutoResetEvent также служит целям синхронизации потоков. Этот класс является оберткой над объектом ОС Windows "событие" и позволяет переключить данный объект-событие из сигнального в несигнальное состояние. Так, пример из предыдущей темы мы можем переписать с использованием AutoResetEvent следующим образом

class Program

{

    static AutoResetEvent waitHandler = new AutoResetEvent(true);

    static int x=0;

    static void Main(string[] args)

    {

        for (int i = 0; i < 5; i++)

        {

            Thread myThread = new Thread(Count);

            myThread.Name = "Поток " + i.ToString();

            myThread.Start();

        }

        Console.ReadLine();

    }

    public static void Count()

    {

        waitHandler.WaitOne();

        x = 1;

        for (int i = 1; i < 9; i++)

        {

            Console.WriteLine("{0}: {1}", Thread.CurrentThread.Name, x);

            x++;

            Thread.Sleep(100);

        }

        waitHandler.Set();

    }

}

## Мьютексы

class Program

{

    static Mutex mutexObj = new Mutex();

    static int x=0;

    static void Main(string[] args)

    {

        for (int i = 0; i < 5; i++)

        {

            Thread myThread = new Thread(Count);

            myThread.Name = "Поток " + i.ToString();

            myThread.Start();

        }

        Console.ReadLine();

    }

    public static void Count()

    {

        mutexObj.WaitOne();

        x = 1;

        for (int i = 1; i < 9; i++)

        {

            Console.WriteLine("{0}: {1}", Thread.CurrentThread.Name, x);

            x++;

            Thread.Sleep(100);

        }

        mutexObj.ReleaseMutex();

    }

}

Мы использовали мьютекс для синхронизации потоков. Однако замечательная черта мьютексов состоит также в том, что они могут также применяться не только внутри одного процесса, но и между процессами. Типичный пример - создание приложения, которое можно запустить только один раз. Создадим подобное приложение

 static void Main(string[] args)

        {

            bool existed;

            // получаем GIUD приложения

            string guid = Marshal.GetTypeLibGuidForAssembly(Assembly.GetExecutingAssembly()).ToString();

            Mutex mutexObj = new Mutex(true, guid, out existed);

            if (existed)

            {

                Console.WriteLine("Приложение работает");

            }

            else

            {

                Console.WriteLine("Приложение уже было запущено. И сейчас оно будет закрыто.");

                Thread.Sleep(3000);

                return;

            }

            Console.ReadLine();

        }

    }

}

## Семафоры

using System;

using System.Threading;

namespace SemaphoreApp

{

    class Program

    {

        static void Main(string[] args)

        {

            for (int i = 1; i < 6; i++)

            {

                Reader reader = new Reader(i);

            }

            Console.ReadLine();

        }

    }

    class Reader

    {

        // создаем семафор

        static Semaphore sem = new Semaphore(3, 3);

        Thread myThread;

        int count = 3;// счетчик чтения

        public Reader(int i)

        {

            myThread = new Thread(Read);

            myThread.Name = "Читатель " + i.ToString();

            myThread.Start();

        }

        public void Read()

        {

            while (count > 0)

            {

                sem.WaitOne();

                Console.WriteLine("{0} входит в библиотеку", Thread.CurrentThread.Name);

                Console.WriteLine("{0} читает", Thread.CurrentThread.Name);

                Thread.Sleep(1000);

                Console.WriteLine("{0} покидает библиотеку", Thread.CurrentThread.Name);

                sem.Release();

                count--;

                Thread.Sleep(1000);

            }

        }

    }

}

Для создания семафора используется класс **Semaphore**: static Semaphore sem = new Semaphore(3, 3);. Его конструктор принимает два параметра: первый указывает, какому числу объектов изначально будет доступен семафор, а второй параметр указывает, какой максимальное число объектов будет использовать данный семафор. В данном случае у нас только три читателя могут одновременно находиться в библиотеке, поэтому максимальное число равно 3.

## Использование таймеров

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

class Program

{

    static void Main(string[] args)

    {

        int num = 0;

        // устанавливаем метод обратного вызова

        TimerCallback tm = new TimerCallback(Count);

        // создаем таймер

        Timer timer = new Timer(tm, num, 0, 2000);

        Console.ReadLine();

    }

    public static void Count(object obj)

    {

        int x = (int)obj;

        for (int i = 1; i < 9; i++, x++)

        {

            Console.WriteLine("{0}", x\*i);

        }

    }

}

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

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

# Параллельное программирование и библиотека TPL

Кроме того, упрощается сама работа по созданию новых потоков. Поэтому начиная с .NET 4.0. рекомендуется использовать именно TPL и ее классы для создания многопоточных приложений, хотя стандартные средства и класс Thread по-прежнему находят широкое применение.

## Задачи и класс Task

  class Program

    {

        static void Main(string[] args)

        {

            Task task = new Task(Display);

            task.Start();

            Console.WriteLine("Выполняется работа метода Main");

            Console.ReadLine();

        }

        static void Display()

        {

            Console.WriteLine("Начало работы метода Display");

            // имитация работы метода

            Thread.Sleep(3000);

            Console.WriteLine("Завершение работы метода Display");

        }

    }

Однако в данном случае у нас есть небольшая проблема: если основной код в методе Main уже отработал, а задача еще не завершилась, то мы можем нажать любую клавишу, и приложение завершит свою работу, вне зависимости, завершилась ли задача или нет. И чтобы указать, что метод Main должен подождать до конца выполнения задачи, нам надо использовать метод Wait

static void Main(string[] args)

{

    Task task = new Task(Display);

    task.Start();

    Console.WriteLine("Выполняется работа метода Main");

    task.Wait();

    Console.ReadLine();

}

static void Main(string[] args)

        {

            // первый массив задач

            Task[] tasks = new Task[5];

            for (int i = 0; i < tasks.Length;i++ )

            {

                tasks[i] = Task.Factory.StartNew(Display);

            }

            // второй массив задач

            Task[] tasks2 = new Task[5];

            for (int i = 0; i < tasks2.Length; i++)

            {

                tasks2[i] = Task.Factory.StartNew(()=>{

                    Console.WriteLine("Задача из лямбда-выражения");

                });

            }

            Console.WriteLine("Выполняется работа метода Main");

            Task.WaitAll(tasks);

            Task.WaitAll(tasks2);

            Console.ReadLine();

        }

        static void Display()

        {

            Console.WriteLine("Начало работы метода Display");

            Thread.Sleep(3000);

            Console.WriteLine("Завершение работы метода Display");

        }

    }

## Работа с классом Task

static void Main(string[] args)

{

    Task task1 = new Task(()=>DisplayMessage("вызов метода с параметрами"));

    task1.Start();

    Task task2 = new Task(Display);

    task2.Start();

    Task task3 = new Task(() =>

    {

        Console.WriteLine("Id задачи: {0}", Task.CurrentId);

    });

    task3.Start();

    Task task4 = Task.Factory.StartNew(() =>

    {

        Console.WriteLine("Id задачи: {0}", Task.CurrentId);

    });

    TaskFactory tf = new TaskFactory();

    Task t5 = tf.StartNew(Display);

    Console.ReadLine();

}

static void Display()

{

    Console.WriteLine("Id задачи: {0}", Task.CurrentId);

}

static void DisplayMessage(string message)

{

    Console.WriteLine("Сообщение: {0}", message);

    Console.WriteLine("Id задачи: {0}", Task.CurrentId);

}

Задачи task1, task2 и task3 используют версию конструктора, которая в качестве параметра принимает делегат. Поэтому задачи task1 и task3 создаются с помощью лямбда-выражения, в которое передается вызов метода с параметром. А для задачи task2 в конструктор напрямую передается название выполняемого метода.

При создании задачи с помощью фабрики Task.Factory мы также передаем в метод Task.Factory.StartNew название метода или лямбда-выражение, только в данном случае метод StartNew сразу же и запускает задачу, поэтому нам не надо делать вызов task4.Start

Последний пример задачи аналогичен предыдущему, только использует класс фабрики задач **TaskFactory**

### Свойства класса Task

Класс Task имеет ряд свойств, с помощью которых мы можем получить информацию об объекте. Некоторые из них:

* **AsyncState**: возвращает объект состояния задачи
* **CurrentId**: возвращает идентификатор текущей задачи
* **Exception**: возвращает объект исключения, возникшего при выполнении задачи
* **Status**: возвращает статус задачи

### Возвращение результатов из задач

Задачи могут не только выполняться как процедуры, но и возвращать определенные результаты.

class Program

{

    static void Main(string[] args)

    {

        Task<int> task1 = new Task<int>(()=>Factorial(5));

        task1.Start();

        Console.WriteLine("Факториал числа 5 равен {0}", task1.Result);

        Task<Book> task2 = new Task<Book>(() =>

        {

            return new Book { Title = "Война и мир", Author = "Л. Толстой" };

        });

        task2.Start();

        Book b = task2.Result;

        Console.WriteLine("Название книги: {0}, автор: {1}", b.Title, b.Author);

        Console.ReadLine();

    }

    static int Factorial(int x)

    {

        int result = 1;

        for (int i = 1; i <= x; i++)

        {

            result \*= i;

        }

        return result;

    }

}

public class Book

{

    public string Title { get; set; }

    public string Author { get; set; }

}

Во-первых, чтобы задать возвращаемый из задачи тип объекта, мы должны типизировать Task. Например, Task<int> - в данном случае задача будет возвращать объект int.

И, во-вторых, в качестве задачи должен выполняться метод, возвращающий данный тип объекта. Например, в первом случае у нас в качестве задачи выполняется функция Factorial, которая принимает числовой параметр и также на выходе возвращает число.

Возвращаемое число будет храниться в свойстве Result: task1.Result. Нам не надо его приводить к типу int, оно уже само по себе будет представлять число.

То же самое и со второй задачей task2. В этом случае в лямбда-выражении возвращается объект Book. И также мы его получаем с помощью task2.Result

## Задачи продолжения

 class Program

    {

        static void Main(string[] args)

        {

            Task task1 = new Task(()=>{

                    Console.WriteLine("Id задачи: {0}", Task.CurrentId);

                });

            // задача продолжения

            Task task2 = task1.ContinueWith(Display);

            task1.Start();

            // ждем окончания второй задачи

            task2.Wait();

            Console.WriteLine("Выполняется работа метода Main");

            Console.ReadLine();

        }

        static void Display(Task t)

        {

            Console.WriteLine("Id задачи: {0}", Task.CurrentId);

            Console.WriteLine("Id предыдущей задачи: {0}", t.Id);

            Thread.Sleep(3000);

        }

    }

Подобным образом мы можем построить целую цепочку последовательно выполняющихся задач.

static void Main(string[] args)

{

    Task task1 = new Task(()=>{

        Console.WriteLine("Id задачи: {0}", Task.CurrentId);

    });

    // задача продолжения

    Task task2 = task1.ContinueWith(Display);

    Task task3 = task1.ContinueWith((Task t) =>

    {

        Console.WriteLine("Id задачи: {0}", Task.CurrentId);

    });

    Task task4 = task2.ContinueWith((Task t) =>

    {

        Console.WriteLine("Id задачи: {0}", Task.CurrentId);

    });

    task1.Start();

    Console.ReadLine();

}

## Класс Parallel

Класс Parallel также является частью TPL и предназначен для упрощения параллельного выполнения кода. Parallel имеет ряд методов, которые позволяют распараллелить задачу.

Одним из методов, позволяющих параллельное выполнение задач, является метод Invoke

static void Main(string[] args)

{

    Parallel.Invoke(Display,

        () => {

                    Console.WriteLine("Выполняется задача {0}", Task.CurrentId);

                    Thread.Sleep(3000);

                },

        () => Factorial(5));

    Console.ReadLine();

}

static void Display()

{

    Console.WriteLine("Выполняется задача {0}", Task.CurrentId);

    Thread.Sleep(3000);

}

static void Factorial(int x)

{

    int result = 1;

    for (int i = 1; i <= x; i++)

    {

        result \*= i;

    }

    Console.WriteLine("Выполняется задача {0}", Task.CurrentId);

    Thread.Sleep(3000);

    Console.WriteLine("Результат {0}", result);

}

Метод Parallel.Invoke в качестве параметра принимает массив объектов Action, то есть мы можем передать в данный метод набор методов, которые будут вызываться при его выполнении. Количество методов может быть различным, но в данном случае мы определяем выполнение трех методов. Опять же как и в случае с классом Task мы можем передать либо название метода, либо лямбда-выражение.

И таким образом, при наличии нескольких ядер на целевой машине данные методы будут выполняться параллельно на различных ядрах

Метод Parallel.For позволяет выполнять итерации цикла параллельно. Он имеет следующее определение: For(int, int, Action<int>), где первый параметр задает начальный индекс элемента в цикле, а второй параметр - конечный индекс. Третий параметр - делегат Action - указывает на метод, который будет выполняться один раз за итерацию

## Отмена задач и параллельных операций. CancellationToken

Параллельное выполнение задач может занимать много времени. И иногда может возникнуть необходимость прервать выполняемую задачу. Для этого .NET предоставляет класс CancellationToken.

static void Main(string[] args)

{

    CancellationTokenSource cancelTokenSource = new CancellationTokenSource();

    CancellationToken token = cancelTokenSource.Token;

    int number = 6;

    Task task1 = new Task(() =>

    {

        int result = 1;

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

        {

            if (token.IsCancellationRequested)

            {

                Console.WriteLine("Операция прервана");

                return;

            }

            result \*= i;

            Console.WriteLine("Факториал числа {0} равен {1}", i, result);

            Thread.Sleep(5000);

        }

    });

    task1.Start();

    Console.WriteLine("Введите Y для отмены операции или другой символ для ее продолжения:");

    string s = Console.ReadLine();

    if (s == "Y")

        cancelTokenSource.Cancel();

    Console.Read();

}

Если операция представляет внешний метод, то ему надо передавать в качестве одного из параметров токен.

 Task task1 = new Task(() => Factorial(5, token));

# Aсинхронное программирование

Также асинхронность несет выгоды в веб-приложениях при обработке запросов от пользователей, при обращении к базам данных или сетевым ресурсам. При больших запросах к базе данных асинхронный метод просто уснет на время, пока не получит данные от БД, а основной поток сможет продолжить свою работу. В синхронном же приложении, если бы код получения данных находился в основном потоке, этот поток просто бы блокировался на время получения данных.

Начиная с .NET 4.5 была изменена концепция создания асинхронных вызовов. Новая модель асинхронных вызовов называется **Task-based Asynchronous Pattern** или сокращенно **TAP**. То есть новая модель основывается на использовании задач Task. Но прежде чем перейти к нововой модели, рассмотрим, как происходили вызовы до .NET 4.5 с помощью асинхронных делегатов

## Асинхронные делегаты

 class Program

    {

        public delegate int DisplayHandler();

        static void Main(string[] args)

        {

            DisplayHandler handler = new DisplayHandler(Display);

            IAsyncResult resultObj = handler.BeginInvoke(null, null);

            Console.WriteLine("Продолжается работа метода Main");

            int result = handler.EndInvoke(resultObj);

            Console.WriteLine("Результат равен {0}", result);

            Console.ReadLine();

        }

        static int Display()

        {

            Console.WriteLine("Начинается работа метода Display....");

            int result = 0;

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

            {

                result += i \* i;

            }

            Thread.Sleep(3000);

            Console.WriteLine("Завершается работа метода Display....");

            return result;

        }

    }

Таким образом, после вызова метода Display через выражение handler.BeginInvoke(null, null) работа метода Main не приостанавливается. А выполнение метода Display через делегат DisplayHandler происходит в другом потоке. И лишь когда выполнение в методе Main дойдет до строки int result = handler.EndInvoke(resultObj); он блокируется и ожидает завершения выполнения метода Display

## IAsyncResult и методы BeginInvoke/EndInvoke

DisplayHandler handler = new DisplayHandler(Display);

IAsyncResult resultObj = handler.BeginInvoke(null, null);

int result = handler.EndInvoke(resultObj);

Метод BeginInvoke принимает, как минимум, два параметра. Первый параметр представляет делегат System.AsyncCallback. AsyncCallback указывает на метод, который будет выполняться в результате завершения работы асинхронного делегата. В данном случае мы его не используем и на его место передаем значение null.

Второй параметр представляет произвольный объект, с помощью которого мы можем передать дополнительную информацию в метод завершения, указанный в предыдущем параметре.

public delegate int DisplayHandler(int k);

    class Program

    {

        static void Main(string[] args)

        {

            DisplayHandler handler = new DisplayHandler(Display);

            IAsyncResult resultObj = handler.BeginInvoke(10, new AsyncCallback(AsyncCompleted), "Асинхронные вызовы");

            Console.WriteLine("Продолжается работа метода Main");

            int res = handler.EndInvoke(resultObj);

            Console.WriteLine("Результат: {0}", res);

            Console.ReadLine();

        }

        static int Display(int k)

        {

            Console.WriteLine("Начинается работа метода Display....");

            int result = 0;

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

            {

                result += k \* i;

            }

            Thread.Sleep(3000);

            Console.WriteLine("Завершается работа метода Display....");

           return result;

        }

        static void AsyncCompleted(IAsyncResult resObj)

        {

            string mes = (string)resObj.AsyncState;

            Console.WriteLine(mes);

            Console.WriteLine("Работа асинхронного делегата завершена");

        }

    }

}

Поскольку наш метод теперь принимает один параметр типа int, то нам надо его передать в методе BeginInvoke в самом начале списка параметров: handler.BeginInvoke(10, new AsyncCallback(AsyncCompleted), "Асинхронные вызовы"). Число 10 как раз будет передаваться в качестве значения параметра k в методе int Display(int k).

Если бы у нас делегат и метод Display принимали бы два параметра, например, int Display(int k, double f), тогда в методе BeginInvoke мы должны бы были указать все значения для них в начале списка параметров, например, handler.BeginInvoke(10, 4.5, new AsyncCallback(AsyncCompleted), "Асинхронные вызовы")

Кроме того, мы также передаем в метод BeginInvoke объект делегата **AsyncCallback**. Этот делегат указывает на метод, срабатывающий после завершения асинхронного вызова. В данном случае это метод AsyncCompleted. Этот метод должен принимать параметр IAsyncResult.

И в качестве третьего параметра передается произвольный объект, хранящий данные о состоянии вызова. В данном случае это обычная строка "Асинхронные вызовы". Хотя это может быть любой объект.

Этот объект затем мы можем получить в методе AsyncCompleted:

string mes = (string)resObj.AsyncState;

## Асинхронные методы, async и await

В .NET 4.5 во фреймворк были добавлены два новых ключевых слова async и await, цель которых - упростить написание асинхронного кода. Вместе с функциональностью задач Task они составляют основу новой модели асинхронного программирования в .NET, которая называется **Task-based Asynchronous Pattern.**

Операторы async и await используются вместе для создания асинхронного метода. Такой метод, определенный с помощью модификатора async и содержащий одно или несколько выражений await, называется **асинхронным методом**.

Ключевое слово **async** указывает, что метод или лямбда-выражение являются асинхронными. А оператор **await** применяется к задаче в асинхронных методах, чтобы приостановить выполнение метода до тех пор, пока эта задача не завершится. При этом выполнение потока, в котором был вызван асинхронный метод, не прерывается.

class Program

{

    static void Main(string[] args)

    {

        DisplayResultAsync();

        Console.ReadLine();

    }

    static async void DisplayResultAsync()

    {

        int num = 5;

        int result = await FactorialAsync(num);

        Thread.Sleep(3000);

        Console.WriteLine("Факториал числа {0} равен {1}", num, result);

    }

    static Task<int> FactorialAsync(int x)

    {

        int result = 1;

        return Task.Run(() =>

        {

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

            {

                result \*= i;

            }

            return result;

        });

    }

}

И в методе Main мы вызываем этот асинхронный метод. Однако, как говорилось в теме про класс Task, если у нас асинхронная задача выполняется довольно приличное время, то мы можем не дождаться окончания ее выполнения и также щелкнуть на любую клавишу и выйти из программы. Чтобы в обязательном порядке дождаться окончания выполнения, нужно определить метод DisplayResultAsync()как задачу и у нее вызвать метод Wait

static void Main(string[] args)

{

    Task t = DisplayResultAsync();

    t.Wait();

    Console.ReadLine();

}

static async Task DisplayResultAsync()

{

    int num = 5;

    int result = await FactorialAsync(num);

    Thread.Sleep(3000);

    Console.WriteLine("Факториал числа {0} равен {1}", num, result);

}

Либо можно применить методы .GetAwaiter(), который остановит выполнение в вызывающем потоке, пока задача не завершится, и .GetResult(), который возвращает результат:

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | static void Main(string[] args)  {      DisplayResultAsync().GetAwaiter().GetResult();      Console.WriteLine("Задача DisplayResultAsync завершена");      Console.ReadLine();  } |

Также в данном примере мы можем также определить в качестве асинхронного метода и метод подсчета факториала:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13 | static async Task<int> FactorialAsync(int x)  {      int result = 1;        return await Task.Run(() =>      {          for (int i = 1; i <= x; i++)          {              result \*= i;          }          return result;      });  } |

При определении асинхронных методов надо учитывать, что в качестве возвращаемого результата должен быть объект Task<T> или просто Task (если метод не возвращает результата, как DisplayResultAsync) или void

## Последовательный и параллельный вызов асинхронных методов

Однако не всегда существует подобная зависимость между задачами. В этом случае мы можем запустить все задачи одновременно через метод **Task.WhenAll**:

static async Task DisplayResultAsync()

{

    int num1 = 5;

    int num2 = 6;

    Task<int> t1 = FactorialAsync(num1);

    Task<int> t2 = FactorialAsync(num2);

    Task<int> t3 = Task.Run(() =>

    {

        int res = 1;

        for (int i = 1; i <= 9; i++)

        {

            res += i \* i;

        }

        return res;

    });

    await Task.WhenAll(new[] { t1, t2, t3 });

    Console.WriteLine("Факториал числа {0} равен {1}", num1, t1.Result);

    Console.WriteLine("Факториал числа {0} равен {1}", num2, t2.Result);

    Console.WriteLine("Сумма квадратов чисел равна {0}", t3.Result);

}

В этом случае мы можем запустить все задачи одновременно (например как здесь - через вызовы **Task.Run()**).  
Далее с помощью статического метода **Task.WhenAll()** мы создаем еще одну специальную задачу, автоматически выполняющуюся когда выполнятся все переданные ей задачи. И с помощью оператора **await** ожидаем завершения этой специальной задачи.

## Обработка ошибок в асинхронных методах

class Program

    {

        static void Main(string[] args)

        {

            DisplayResultAsync(0).Wait();

            Console.ReadLine();

        }

        static async Task DisplayResultAsync(int num)

        {

            try

            {

                int result = await Factorial(num);

                Thread.Sleep(3000);

                Console.WriteLine("Факториал числа {0} равен {1}", num, result);

            }

            catch (Exception ex)

            {

                Console.WriteLine(ex.Message);

            }

        }

        static async Task<int> Factorial(int x)

        {

            int result = 1;

            if (x < 1)

                throw new Exception("Число не должно быть меньше 1");

            return await Task.Run(() =>

            {

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

                {

                    result \*= i;

                }

                return result;

            });

        }

    }

}

### await в блоках catch и finally

Начиная с версии C# 6.0 в язык была добавлена возможность вызова асинхронного кода в блоках catch и finally. Так, возьмем предыдущий пример с подсчетом факториала:

static void Main(string[] args)

{

    DisplayResultAsync(0).GetAwaiter();

    Console.Read();

}

static async Task DisplayResultAsync(int num)

{

    try

    {

        int result = 1;

        if (num < 1)

            throw new Exception("Число не должно быть меньше 1");

        result= await Task.Run(() =>

        {

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

            {

                result \*= i;

            }

            return result;

        });

        Console.WriteLine("Факториал числа {0} равен {1}", num, result);

    }

    catch (Exception ex)

    {

        await Log(ex);

    }

    finally

    {

        await Task.Run(() => Console.WriteLine("await в блоке finally"));

    }

}

static async Task Log(Exception ex)

{

    await Task.Run(() =>

    {

        Console.WriteLine(ex);

    });

}

## Отмена асинхронных операций

Для отмены асинхронных операций используются классы CancellationToken и CancellationTokenSource.

static void Main(string[] args)

{

    DisplayResultAsync(6);

     Console.Read();

}

static async void DisplayResultAsync(int num)

{

    CancellationTokenSource cts = new CancellationTokenSource();

    try

    {

        Task t1 = FactorialAsync(num, cts.Token);

        Task t2 = Task.Run(() =>

        {

            Thread.Sleep(2000);

            cts.Cancel(); // отмена асинхронной операции

        });

        await Task.WhenAll(t1, t2);

    }

    catch(OperationCanceledException ex)

    {

        Console.WriteLine(ex.Message);

    }

    finally

    {

        cts.Dispose();

    }

}

static Task FactorialAsync(int x, CancellationToken token)

{

    return Task.Run(() =>

    {

        int result = 1;

        for (int i = 1; i <= x; i++)

        {

            token.ThrowIfCancellationRequested();

            result \*= i;

            Console.WriteLine("Факториал числа {0} равен {1}", i, result);

            Thread.Sleep(1000);

        }

    }, token);

}

# Рефлексия

**Рефлексия** представляет собой процесс выявления типов во время выполнения приложения. Каждое приложение содержит набор используемых классов, интерфейсов, а также их методов, свойств и прочих кирпичиков, из которых складывается приложение. И рефлексия как раз и позволяет определить все эти составные элементы приложения

* **Assembly**: класс, представляющий сборку и позволяющий манипулировать этой сборкой
* **AssemblyName**: класс, хранящий информацию о сборке
* **MemberInfo**: базовый абстрактный класс, определяющий общий функционал для классов EventInfo, FieldInfo, MethodInfo и PropertyInfo
* **EventInfo**: класс, хранящий информацию о событии
* **FieldInfo**: хранит информацию об определенном поле типа
* **MethodInfo**: хранит информацию об определенном методе
* **PropertyInfo**: хранит информацию о свойстве
* **ConstructorInfo**: класс, представляющий конструктор
* **Module**: класс, позволяющий получить доступ к определенному модулю внутри сборки
* **ParameterInfo**: класс, хранящий информацию о параметре метода
* class Program
* {
* static void Main(string[] args)
* {
* Type myType = typeof(User);
* Console.WriteLine(myType.ToString());
* Console.ReadLine();
* }
* }
* public class User
* {
* public string Name { get; set; }
* public int Age { get; set; }
* public User(string n, int a)
* {
* Name = n;
* Age = a;
* }
* public void Display()
* {
* Console.WriteLine("Имя: {0}  Возраст: {1}", this.Name, this.Age);
* }
* public int Payment(int hours, int perhour)
* {
* return hours \* perhour;
* }
* }
* Применим метод GetMembers и выведем всю информацию о типе:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11 | static void Main(string[] args)  {      Type myType = Type.GetType("TestConsole.User", false, true);        foreach(MemberInfo mi in myType.GetMembers())      {          Console.WriteLine(mi.DeclaringType + " " + mi.MemberType + " " + mi.Name);      }        Console.ReadLine();  } |