Приднестровский государственный университет им. Т.Г. Шевченко

физико-математический факультет

кафедра прикладной математики и информатики

Л А Б О Р А Т О Р Н А Я Р А Б О Т А № 8

по дисциплине:
«*Системы программирования*»

Тема:
«Обработка текстовых данных»

РАЗРАБОТАЛИ:
ст. преподаватель кафедры ПМиИ
Великодный В.И.
ст. преподаватель кафедры ПМиИ
Калинкова Е.В.

г. Тирасполь

Цель работы:

Изучить основные методы для работы со строками. Закрепить полученные знания при составлении программ на обработку строк при решении задач на языке C#.

Теоретическая часть

Строки и класс System.String

В языке C# строковые значения представляет тип `string`, а вся функциональность работы с данным типом сосредоточена в классе `System.String`. Собственно, `string` является псевдонимом для класса `System.String`. Объект этого класса представляет текст как последовательность символов Unicode.

Важно заметить, что строки типа `string` в целях оптимизации сделаны неизменяемыми. То есть, чтобы изменить какой-то символ в строке нужно создавать её копию с другим символом в требуемой позиции. Если нужно часто добавлять к строке новые символы, править её или удалять какие-то фрагменты, то лучше использовать класс `System.Text.StringBuilder`.

Создавать строки можно, как используя переменную типа string и присваивая ей значение, так и применяя один из конструкторов класса `String`:

```
string s1 = "hello";
string s2 = new string('a', 6); // результат: строка "aaaaaa"
```

Объект типа `string` можно также создать из массива типа `char`. Например:

```
char[] chararray = {'w', 'o', 'r', 'l', 'd'};
string s3 = new string(chararray);
```

или

```
string s3 = new string(new []{'w', 'o', 'r', 'l', 'd'});
```

Мы можем обратиться к строке как к массиву символов и получить по индексу любой из ее символов:

```
string s1 = "hello";
char ch1 = s1[1]; // символ 'e'
Console.WriteLine(ch1);
```

Используя свойство `Length`, как и в обычном массиве, можно получить длину строки.

```
Console.WriteLine(s1.Length);
```

Строку можно сформировать по шаблону с помощью метода `Format`. Его работа аналогична работе `Console.Write`, но результат не выводится на экран, а возвращается методом.

```
string world = "мир";
string text = string.Format("Привет, {0}!", world);
```

Работа со строками

Конкатенация

Конкатенация строк или объединение может производиться как с помощью операции +, так и с помощью метода `Concat`:

```
string s1 = "hello";
string s2 = "world";
string s3 = s1 + " " + s2; // результат: строка "hello world"
string s4 = string.Concat(s3, "!!!"); // результат: строка "hello world!!!"
```

Метод `Concat` является статическим методом класса `String`, принимающим в качестве параметров две строки. Также имеются другие версии метода, принимающие другое количество параметров.

Для объединения строк также может использоваться метод `Join`:

```
string s5 = "кукушка";
string s6 = "кукушонку";
string s7 = "купила";
string s8 = "капюшон";
string[] values = new string[] { s5, s6, s7, s8 };
string s9 = string.Join(" ", values);
// результат: строка "кукушка кукушонку купила капюшон"

string s10 = "один";
string s11 = "два";
string s12 = "три";
string[] numbers = new string[] { s10, s11, s12 };
string s13 = string.Join(", ", numbers); // результат: строка "один, два, три"
```

Метод `Join` также является статическим. Использованная выше версия метода получает два параметра: строку-разделитель (в первом примере пробел, во втором примере запятая и пробел) и массив строк, которые будут соединяться и разделяться разделителем.

Метод `Join` может аналогично использоваться для объединения элементов произвольных коллекций (для каждого элемента будет вызван метод `ToString()`).

```
string s14 = string.Join(" ", new[] { 1, 2, 3 }); // результат: строка "1 2 3"
```

Сравнение строк

Для сравнения строк применяется статический метод `Compare`. Строки сравниваются лексикографически, то есть та строка меньше, которая стояла бы в словаре раньше. Строка `"a"` "меньше" строки `"b"`, `"bb"` "больше" строки `"ba"`. Если обе строки равны – метод возвращает "0", если первая строка меньше второй – "-1", если первая больше второй – "1":

```
string.Compare("абрикос", "вишня"); // возвращает -1
string.Compare("весна", "весло");   // возвращает 1
string.Compare("котик", "кот");     // возвращает 1
string.Compare("пес", "пес");       // возвращает 0
```

Чтобы игнорировать регистр букв, в метод нужно передать третий аргумент `true`.

```
string.Compare("кот", "Кот"); // возвращает -1
string.Compare("кот", "Кот", true); // возвращает 0
```

Поиск в строке

С помощью метода `IndexOf` мы можем определить индекс первого вхождения отдельного символа или подстроки в строке:

```
string s1 = "hello world";
char ch = 'o';
int index1 = s1.IndexOf(ch); // равно 4
string s2 = "wo";
int index2 = s1.IndexOf(s2); // равно 6
```

Подобным образом действует метод `LastIndexOf`, только находит индекс последнего вхождения символа или подстроки в строку.

Еще одна группа методов позволяет узнать начинается или заканчивается строка на определенную подстроку. Для этого предназначены методы `StartsWith` и `EndsWith`. Метод `Contains` позволяет определить, содержится ли в строке определенная подстрока.

```
string s3 = "колокол";
```

```
Console.WriteLine(s3.StartsWith("кол"));        // результат true
Console.WriteLine(s3.EndsWith("кол"));          // результат true
Console.WriteLine(s3.EndsWith("ко"));           // результат false
Console.WriteLine(s3.Contains("око"));          // результат true
Console.WriteLine(s3.Contains("молоко"));       // результат false
```

Получение подстроки из строки

Извлечь подстроку из строки позволяет метод `Substring`:

```
string text = "протокол";
string word1 = text.Substring(5);         // результат "кол"
string word2 = text.Substring(4, 3);      // результат "око"
```

В первом примере метода `Substring` извлекается подстрока, начиная с места, обозначаемого параметром, и до конца вызывающей строки. А во втором примере извлекается подстрока, состоящая из количества символов, определяемых вторым параметром, начиная с места, обозначаемого первым параметром.

Разделение строк

С помощью метода `Split` мы можем разделить строку на массив подстрок. В качестве параметра функция `Split` принимает массив символов или строк, которые и будут служить разделителями. Например, подсчитаем количество слов в строке, разделив ее по пробельным символам:

```
string text = "Мороз и солнце";
string[] words = text.Split(' ');
Console.WriteLine(words.Length); // результат 3
```

В следующем примере используем массив разделителей:

```
string text = "один,два,три и четыре";
string[] words = text.Split(new char[] {' ', ','});
Console.WriteLine(words.Length); // результат 5
```

Это не лучший способ разделения по пробелам, так как во входной строке у нас могло бы быть несколько подряд идущих пробелов и в итоговый массив также бы попадали пробелы, поэтому лучше использовать другую версию метода:

```
string text = " один, два,  три, четыре ";
string[] words = text.Split(new char[] {' ',','},
StringSplitOptions.RemoveEmptyEntries);
Console.WriteLine(words.Length); // результат 4
```

Второй параметр `StringSplitOptions.RemoveEmptyEntries` говорит, что надо удалить все пустые подстроки.

Обрезка строки

Для обрезки начальных или концевых символов используется метод `Trim`:

```
string text = " hello world ";
text = text.Trim(); // результат "hello world"
text = text.Trim(new char[] { 'd', 'h' }); // результат "ello worl"
```

Метод `Trim` без параметров обрезает начальные и конечные пробелы и возвращает обрезанную строку. Чтобы явным образом указать, какие начальные и конечные символы следует обрезать, мы можем передать в метод массив этих символов.

Эта метод имеет частичные аналоги: метод `TrimStart` обрезает начальные символы, а метод `TrimEnd` обрезает конечные символы.

Вставка строк

Для вставки одной строки в другую применяется метод `Insert`:

```
string text = "Хороший день";
string word = "летний ";
text = text.Insert(8, word); // результат: "Хороший летний день"
```

Первым параметром в методе `Insert` является индекс, по которому надо вставлять подстроку, а второй параметр – собственно подстрока.

Удаление строк

Удалить часть строки помогает метод `Remove`:

```
string text = "информатика";
int ind = text.Length - 1; // индекс последнего символа
// вырезаем последний символ
text = text.Remove(ind);  // результат: "информатик"
// вырезаем первые два символа
text = text.Remove(0, 2); // результат: "форматик"
// вырезаем последние три символа
text = text.Remove(text.Length - 3); // результат: "форма"
```

Первая версия метода `Remove` принимает индекс в строке, начиная с которого надо удалить все символы. Вторая версия принимает еще один параметр – сколько символов надо удалить.

Замена

Чтобы заменить один символ или подстроку на другую, применяется метод `Replace`:

```
string text = "хороший день";
text = text.Replace("хороший", "морозный"); // результат: "морозный день"
text = text.Replace("о", "");               // результат: "мрзный день"
```

Во втором случае применения метода `Replace` строка из одного символа "о" заменяется на пустую строку, то есть фактически удаляется из текста. Подобным способом легко удалять какой-то определенный текст в строках.

Смена регистра

Для приведения строки к верхнему и нижнему регистру используются соответственно методы `ToUpper()` и `ToLower()`:

```
string text = "Hello world!";
Console.WriteLine(text.ToLower()); // hello world!
Console.WriteLine(text.ToUpper()); // HELLO WORLD!
```

Класс StringBuilder

Объекты класса `string` представляют собой неизменяемые (`immutable`) последовательности символов Unicode. Когда вы используете любой метод по изменению строки (например, `Replace()`), он возвращает новую измененную копию строки, исходные же строки остаются неизменными. Так сделано потому, что операция создания новой строки гораздо менее затратна, чем операции копирования и сравнения, что повышает скорость работы программы. В C# также есть класс `StringBuilder`, который позволяет изменять строки. Этот класс находится в пространстве имён `System.Text`.

Рассмотрим работу с ним на примерах.

```
StringBuilder sb0 = new StringBuilder();  // Пустая строка
StringBuilder sb = new StringBuilder("Привет"); // Создание с инициализацией
sb.Append(", ");                          // Добавить текст в конец
```

```
// Добавить в конец с форматированием
string world = "мир";
sb.AppendFormat("{0}?", world); // sb: "Привет, мир?"
// Заменить последний символ
sb[sb.Length - 1] = '!';        // sb: "Привет, мир!"
```

Регулярные выражения

Регулярное выражение **–** это шаблон, по которому выполняется поиск соответствующего фрагмента текста.

Регулярные выражения предоставляют массу возможностей. Некоторые из них:

- заменять в строке все одинаковые слова другим словом, или удалять такие слова;

- выделять из строки необходимую часть. Например, из любой ссылки выделять только доменную часть;

- проверять соответствует ли строка заданному шаблону. Например, проверять, правильно ли введен email, телефон т.д.;

- проверять, содержит ли строка заданную подстроку;

- извлекать из строки все вхождения подстрок, соответствующие шаблону регулярного выражения. Например, получить все даты из строки.

Метасимволы в регулярных выражениях

Язык описания регулярных выражений состоит из символов двух видов: обычных символов и метасимволов. Обычный символ представляет в выражении сам себя, а метасимвол – некоторый *класс символов*.

Рассмотрим наиболее употребительные метасимволы:

Класс символов	Описание	Пример
`.`	Любой символ, кроме \n.	Выражение `c.t` соответствует фрагментам: `cat`, `cut`, `c#t`, `c{t` и т.д.
`[]`	Любой одиночный символ из последовательности, записанной внутри скобок. Допускается использование диапазонов символов.	Выражение `c[aui]t` соответствует фрагментам: `cat`, `cut`, `cit`. Выражение `c[a-c]t` соответствует фрагментам: `cat`, `cbt`, `cct`.
`[^]`	Любой одиночный символ, не входящий в последовательность, записанную внутри скобок. Допускается использование диапазонов символов.	Выражение `c[^aui]t` соответствует фрагментам: `cbt`, `cct`, `c2t` и т.д. Выражение `c[^a-c]t` соответствует фрагментам: `cdt`, `cet`, `c%t` и т.д.
`\w`	Любой алфавитно-цифровой символ.	Выражение `c\wt` соответствует фрагментам: `cbt`, `cct`, `c2t` и т.д., но не соответствует фрагментам `c%t`, `c{t` и т.д.
`\W`	Любой не алфавитно-цифровой символ.	Выражение `c\Wt` соответствует фрагментам: `c%t`, `c{t`, `c.t` и т.д., но не соответствует фрагментам `cbt`, `cct`, `c2t` и т.д.
`\s`	Любой пробельный символ.	Выражение `\s\w\w\w\s` соответствует любому слову из трех букв, окруженному пробельными символами.
`\S`	Любой непробельный символ.	Выражение `\s\S\S\S\s` соответствует любым трем непробельным символам, окруженным пробельными.
`\d`	Любая десятичная цифра.	Выражение `c\dt` соответствует фрагментам: `c1t`, `c2t`, `c3t` и т.д.
`\D`	Любой символ, не являющийся десятичной цифрой.	Выражение `c\Dt` не соответствует фрагментам: `c1t`, `c2t`, `c3t` и т.д.

Альтернативы можно перечислять через вертикальную черту. Например, шаблон "(a|b)c" соответствует строкам «ac» и «bc».

Кроме метасимволов, обозначающие классы символов, могут применяться уточняющие метасимволы:

Уточняющие символы	Описание
^	Фрагмент, совпадающий с регулярными выражениями, следует искать только в начале строки
$	Фрагмент, совпадающий с регулярными выражениями, следует искать только в конце строки
\A	Фрагмент, совпадающий с регулярными выражениями, следует искать только в начале многострочной строки
\Z	Фрагмент, совпадающий с регулярными выражениями, следует искать только в конце многострочной строки
\b	Фрагмент, совпадающий с регулярными выражениями, начинается или заканчивается на границе слова, т.е. между символами, соответствующими метасимволам \w и \W
\B	Фрагмент, совпадающий с регулярными выражениями, не должен встречаться на границе слов

Например, "^abc" будет соответствовать только строкам, начинающимся с «abc», а не просто содержащих эти буквы.

В регулярных выражениях часто используются повторители – метасимволы, которые располагаются непосредственно после обычного символа или группы символов и задают количество его повторений в выражении.

Повторители	Описание	Пример
*	Ноль или более повторений предыдущего элемента	Выражение ca*t соответствует фрагментам: ct, cat, caat, caaat и т.д.
+	Одно или более повторений предыдущего элемента	Выражение ca+t соответствует фрагментам: cat, caat, caaat и т.д.
?	Не более одного повторения предыдущего элемента	Выражение ca?t соответствует фрагментам: ct, cat.
{n}	Ровно n повторений предыдущего элемента	Выражение ca{3}t соответствует фрагменту: caaat. Выражение (cat){2} соответствует фрагменту: catcat.
{n,}	По крайней мере n повторений предыдущего элемента	Выражение ca{3,}t соответствует фрагментам: caaat, caaaat, caaaaaaat и т.д. Выражение (cat){2,} соответствует фрагментам: catcat, catcatcat и т.д.
{n, m}	От n до m повторений предыдущего элемента.	Выражение ca{2,4}t соответствует фрагментам: caat, caaat, caaaat.

Регулярное выражение записывается в виде строкового литерала, причем перед строкой необходимо ставить символ @, который говорит о том, что строку нужно будет рассматривать и в том случае, если она будет занимать несколько строчек на экране. Однако символ @ можно не ставить, если в качестве шаблона используется шаблон без метасимволов.

Все повторители жадные, то есть стараются захватить как можно больше символов. Например, в строке «abbbaccca» шаблон @"a.*a" будет соответствовать всей строке. Чтобы сделать повторитель нежадным, нужно поставить после него вопросительный знак. Таким образом, «+?» и «*?» – нежадные аналоги «+» и «*». В рассмотренном примере шаблон @"a.*?a" будет соответствовать минимальной удовлетворяющей шаблону подстроке – «abbba».

Замечание. Если нужно найти какой-то символ, который является метасимволом, например, точку, можно это сделать, защитив ее обратным слэшем. Т.е. просто точка означает любой одиночный символ, а \. означает просто точку.

Примеры регулярных выражений:

1. слово rus – `@"rus"` или `"rus"`
2. номер телефона в формате xxx-xx-xx – `@"\d\d\d-\d\d-\d\d"` или `@"\d{3}(-\d\d){2}"`
3. автомобильный номер ПМР – `@"[АВЕКРСТ]\d{3}[АВЕКМНОРСТХ]{2}"`
4. дата в формате ISO (ГГГГ-ММ-ДД, такой способ записи удобен для сортировки) – `"\d\d\d\d-\d\d-\d\d"`

Поиск в тексте по шаблону

Пространство имен библиотеки базовых классов `System.Text.RegularExpressions` содержит все классы платформы .NET Framework, имеющие отношение к регулярным выражениям. Важнейшим классом, поддерживающим их, является класс `Regex`, который представляет неизменяемые откомпилированные регулярные выражения. Для описания регулярного выражения в классе определено несколько перегруженных конструкторов:

1. `Regex()` – создает пустое выражение;
2. `Regex(String)` – создает заданное выражение;
3. `Regex(String, RegexOptions)` – создает заданное выражение и задает параметры для его обработки с помощью элементов перечисления `RegexOptions` (например, различать или нет прописные и строчные буквы).

Поиск фрагментов строки, соответствующих заданному выражению, выполняется с помощью методов `IsMatch`, `Match`, `Matches` класса `Regex`.

Метод `IsMatch` возвращает `true`, если фрагмент, соответствующий выражению, в заданной строке найден, и `false` в противном случае. Например, попытаемся определить, встречается ли в заданном тексте слово `собака`:

```
static void Main()
{
    Regex re = new Regex("собака", RegexOptions.IgnoreCase);
    string text1 = "Кот в доме, собака в конуре.";
    string text2 = "Котик в доме, собачка в конуре.";
    Console.WriteLine(re.IsMatch(text1));       // true
    Console.WriteLine(re.IsMatch(text2));       // false
}
```

Замечание. `RegexOptions.IgnoreCase` означает, что регулярное выражение применяется без учета регистра символов.

Можно использовать конструкцию выбора из нескольких элементов. Варианты выбора перечисляются через вертикальную черту. Например, попытаемся определить, встречается ли в заданном тексте слов `собака` или `кот`:

```
static void Main()
{
    Regex re = new Regex("собака|кот", RegexOptions.IgnoreCase);
    string text1 = "Кот в доме, собака в конуре.";
    string text2 = "Котик в доме, собачка в конуре.";
    Console.WriteLine(re.IsMatch(text1));    // true
    Console.WriteLine(re.IsMatch(text2));    // true
    Console.ReadLine();
}
```

Попытаемся определить, есть ли в заданных строках номера телефона в формате xx-xx-xx или xxx-xx-xx:

```
static void Main()
{
```

```
    Regex re = new Regex(@"\d{2,3}(-\d\d){2}");
    string text1 = "tel:123-45-67";
    string text2 = "tel:no";
    string text3 = "tel:12-34-56";
    Console.WriteLine(re.IsMatch(text1));    // true
    Console.WriteLine(re.IsMatch(text2));    // false
    Console.WriteLine(re.IsMatch(text3));    // true
}
```

Метод `Match` класса `Regex` не просто определяет, содержится ли текст, соответствующий шаблону, а возвращает объект класса `Match`, содержащий первый фрагмент, совпавший с шаблоном. При помощи метода `NextMatch` можно переходить к следующему совпадающему фрагменту. Если фрагмент не найден, то свойство `Success` будет равно `false`.

Следующий пример позволяет найти все номера телефонов в указанном фрагменте текста:

```
static void Main()
{
    Regex re = new Regex(@"\d{2,3}(-\d\d){2}");
    string text = @"Контакты в Москве tel:123-45-67, 123-34-56; fax:123-56-45
                  Контакты в Саратове tel:12-34-56; fax:12-56-45";
    Match tel = re.Match(text);
    while (tel.Success)
    {
        Console.WriteLine(tel);
        tel = tel.NextMatch();
    }
}
```

Следующий пример позволяет вывести целые числа, встречающиеся в тексте:

```
static void Main()
{
    Regex re = new Regex(@"[-+]?\d+");
    string text = @"5*10=50 -80/40=-2";
    Match teg = re.Match(text);
    while (teg.Success)
    {
        Console.WriteLine(teg);
        teg = teg.NextMatch();
    }
}
```

Метод `Matches` класса `Regex` возвращает объект класса `MatchCollection` – коллекцию всех фрагментов заданной строки, совпавших с шаблоном. При этом метод `Matches` многократно запускает метод `Match`, каждый раз начиная поиск с того места, на котором закончился предыдущий поиск.

```
static void Main()
{
    string text = @"5*10=50 -80/40=-2";
    Regex re = new Regex(@"[-+]?\d+");
    MatchCollection matches = re.Matches(text);
    foreach (Match m in matches)
    {
        Console.Write("{0} ", m.ToString());
    }
}
```

Для того чтобы можно было применять `Linq` к коллекции типа `MatchCollection`, её необходимо преобразовать с помощью метода `Cast<Match>`. Например, выберем в предыдущем примере только положительные числа.

```
var r = from m in match.Cast<Match>()
            where int.Parse(m.ToString()) > 0
            select int.Parse(m.ToString());
```

Просмотр вперёд и назад

Часто требуется, чтобы символы слева и справа от шаблона удовлетворяли какому-то условию, которое частью самого шаблона не является. Например, если требуется найти все слова в кавычках, не захватывая при этом сами кавычки. Для решения этой задачи можно использовать специальные группы «(?<=*подшаблон*)» и «(?=*подшаблон*)». Они используются для «заглядывания» назад и вперёд соответственно.

Пример. Дан текст на русском языке. Вывести на экран слова в кавычках-ёлочках. Словом будем считать последовательность букв (без дефиса).

```
string text = "Слова «шкаф» и «адмирал» иностранного происхождения.";
Regex re = new Regex(@"(?<=«)[а-яА-Я]+(?=»)");
foreach (var m in re.Matches(text))
    Console.WriteLine(m);
```

Редактирование текста

Регулярные выражения могут эффективно использоваться для редактирования текста. Например, метод `Replace` класса `Regex` позволяет выполнять замену одного фрагмента текста другим или удаление фрагментов текста. Метод `Split` возвращает массив строк, полученный в результате разделения входящей строки в местах соответствия шаблону регулярного выражения.

Пример. Изменение номеров телефонов:

```
static void Main()
{
    string text = @"Контакты в Москве tel:123-45-67, 123-34-56; fax:123-56-45.
Контакты в Саратове tel:12-34-56; fax:11-56-45";
    Console.WriteLine("Старые данные\n" + text);
    string newText = Regex.Replace(text, "123-", "890-");
    Console.WriteLine("Новые данные\n" + newText);
}
```

Пример. Удаление всех номеров телефонов из текста:

```
static void Main()
{
    string text = @"Контакты в Москве tel:123-45-67, 123-34-56; fax:123-56-45.
Контакты в Саратове tel:12-34-56; fax:12-56-45";
    Console.WriteLine("Старые данные\n" + text);
    string newText = Regex.Replace(text, @"\d{2,3}(-\d\d){2}", "");
    Console.WriteLine("Новые данные\n" + newText);
}
```

Пример. Разбиение исходного текста на фрагменты:

```
static void Main()
{
    string text = @"Контакты в Москве tel:123-45-67, 123-34-56; fax:123-56-45.
Контакты в Саратове tel:12-34-56; fax:12-56-45";
    string[] newText = Regex.Split(text, "[ ,.:;]+");
    foreach (string a in newText)
        Console.WriteLine(a);
}
```

Работа с группами

При поиске или замене можно выделять в тексте не только всё выражение, но и его фрагменты. Для этого фрагменты, называемые группами, должны быть заключены в круглые скобки. Фрагменты нумеруются по порядковому номеру открывающей скобки. Группа 0 – всё выражение.

Например, в выражении, соответствующем времени `"(\d\d):(\d\d)"` в первой группе будут находиться первые две цифры, а во второй – вторые две. Группы могут быть и вложенными.

На группу можно сослаться в самом регулярном выражении с помощью обратной косой черты и номера группы. Например, выражение `@"(\d+)=\1"` соответствует всем парам одинаковых чисел, разделённых знаком «=». Здесь `\1` соответствует тексту, захваченному подвыражением в первых скобках.

К группам можно получить доступ через свойство `Groups` результата, возвращаемого методом `Match` регулярного выражения.

Например, выведем отдельно часы и минуты.

```
Regex re = new Regex(@"(\d\d):(\d\d)");
Match result = re.Match("17:35");
System.Console.WriteLine("{0} ч. {1} мин.", result.Groups[1], result.Groups[2]);
```

На экране появится «17 ч. 35 мин.»

Также группы можно использовать при замене методом `Replace`. Вместо заменяемой строки используется лямбда-функция, которой передаётся объект с результатом поиска, а функция должна вернуть строку, на которую нужно заменить найденный фрагмент.

```
string text = "Текущее время - 17:35";
Regex re = new Regex(@"(\d\d):(\d\d)");
string text2 = re.Replace(text,
       r => string.Format("{0} ч. {1} мин.", r.Groups[1], r.Groups[2]));
System.Console.WriteLine(text2); // Текущее время - 17 ч. 35 мин.
```

Задания для самостоятельной работы

Разработайте приложения для решения задач из сборника задач по программированию согласно вашему варианту.