Chapter 24. Sample Programming Exam – Topic #1

In This Chapter

In this chapter we will look at and offer **solutions to three problems from a sample programming exam**. While solving them we will put into practice the methodology described in the chapter "Methodology of Problem Solving".

Problem 1: Extract Text from HTML Document

We are given HTML file named **Problem1.html**. Write a program, which **removes all HTML tags** and retains only the text inside them. Output should be written into the file **Problem1.txt**.

Sample input file for **Problem1.html**:

```
<html>
<head><title>Welcome to our site!</title></head>
<body>
<center>
<img src="/en/img/logo.gif" width="130" height="70" alt="Logo">
<br><br><br>
<font size="-1"><a href="/index.html">Home</a>
<a href="/contacts.html">Contacts</a>
<a href="/about.html">About</a></font><p>
</center>
</body>
</html>
```

Sample output file for **Problem1.txt**:

```
Welcome to our site!
Home
Contacts
About
```

Inventing an Idea

The first thing that occurs to us as an **idea for solving this problem** is to read sequentially (e.g. line by line) the input file and to remove any tags. It is easily seen that all tags starting with the character "**<**" and end with the character "**>**". This also applies to opening and closing tags. This means that for each line in the file we should remove all substrings starting with "**<**" and ending with "**>**".

Checking the Idea

We have an idea for solving the problem. Whether the idea is correct? First we **should check it**. We can ensure it is correct for the sample input file, and then consider whether there are specific cases where the idea could be incorrect.

We **take a pen and paper** and check by manually whether the idea is correct. We do this by striking out all text substrings that start with the character "**<**" and end with the character "**>**". As we do so, we see that there is only pure text and any tags disappear:

```
<html>
<head><title>Welcome to our site!</title></head>
<body>
<center>
<img src="/en/img/logo.gif" width="130" height="70" alt="Logo">
<br><br><br>
<font size="-1"><a href="/index.html">Home</a>
<a href="/contacts.html">Contacts</a>
<a href="/about.html">About</a></font><p>
</center>
</body>
</html>
```

Now we have to think of some **more special cases**. We do not want to write 200 lines of code and only then think about special case, finding out we have to redesign the entire program. It is important to check the problematic situations now, before we begin writing the code of the solution. We can think of the following special example:

```
<html><body>
Click<a href="info.html">on this
link</a>for more info.<br />
This is<b>bold</b>text.
</body></html>
```

There are two things to consider:

- There are tags containing text that **open and close at separate lines**. Such tags in our example are **<html>**, **<body>** and **<a>**.
- There are tags that contain text and other tags in themselves (**nested tags**). For example **<body>** and **<html>**.

What should be the result of this example? If we directly remove all tags we will get something like this:

```
Clickon this
linkfor more info.
This isboldtext.
```

Or maybe we should follow the rules of the HTML language and get the following result:

```
Click on this link for more info.
This is bold text.
```

There are other options, such as putting each piece of text, which is not a tag, on a new line:

```
Click
on this
link
for more info.
This is
bold
text.
```

If we remove all the text in tags and snap the other text, we will get **words that are stuck together**. From the task's description it is not clear if this is the requested result or it must be as in the HTML language. In the HTML language each series of separators (spaces, new lines, tabs, etc.) appear as a space. However, this was not mentioned in the task's description it is not clear from the sample input and output.

It is not clear yet whether to print the words that are in a tag which holds other tags or to skip them. If we print only the contents of the tags, which consist of text only, we will get something like this:

```
on this
link
bold
```

It is yet not clear from the description, how to display the text that is located on a few lines inside a tag.

Clarification of the Statement of the Problem

The first thing to do when we find ambiguity in the description of the task is to **read it carefully**. In this case the problem statement is not really clear and does not give us the answers. Probably we should not follow the HTML rules, because they are not described in the problem statement, but it is not clear whether to connect the words in neighboring tags or separate them by a space or new line.

This leaves us only one thing – **to ask**. If we have an exam, we will ask the one who gives us the task. In real life, someone is an **owner** of the software we develop, and he could **answer the questions**. If nobody can give an answer, choose one option that seems most correct under the information we have and act on it. Assume that we need to print text, which remains after removing all opening and closing tags, **using a blank line separator at the positions of the tags**. If there are blank lines in the text, we keep them. For our example, we should obtain the following **correct output**:

```
Click
on this
link
for more info.
This is
bold
text.
```

A New Idea for Solving the Problem

So, after adapting these **new requirements**, the following idea comes: read file line by line and **substitute each tag with a new line**. To avoid duplication of new lines in the resulting file, replace every two consecutive lines of new results with a new line.

We **check the new idea** with the example from the original statement of the problem with our example to ensure it is correct. It remains to implement it.

Break a Task into Subtasks

The task can easily break into **3 subtasks**:

- **Read** the input file.
- **Processing** of a line of input file: replace tags with a new line.
- **Print** results in the output file.

What Data Structures to Use?

In this task we must perform simple word processing and file management. The question of what data structures to use is not a problem – **for word processing** we use **`string`** and if necessary – **`StringBuilder`**.

Consider the Efficiency

If we read the lines one by one, it will not be a slow operation. Processing of one line can be done by replacing some characters with others – a quick operation. We should not have performance problems.

A possible problem could be the clearing of the empty lines. If we collect all lines in a buffer **(`StringBuilder`)** and then remove double blank lines, this buffer will occupy too much memory for large input files (for example 500 MB input file).

To save memory, we will try to clean the excess blank lines just after the replacement tags with the white space character.

Now we examined the idea of solving the task, we ensured that it is good and covers the special cases that may arise, and believe **we will have no performance issues**.

Now we can safely **proceed to implementation** of the algorithm. We will write the code step by step to find errors as early as possible.

Step 1 – Read the Input File

The first step solving the given task is **reading the input file**. In our case it is a HTML file. This should not bother us, because HTML is a text format. Therefore, to read it, we will use the **`StreamReader`** class. We will traverse the input file line by line and each line we will derive (for now it is not important how we will do it) all the information we need (if any) and save it into an object of type **`StringBuilder`**. Extraction we will implement in the next step (step 2). Let's write the necessary code for the implementation of our first step:

```
string line = string.Empty;
StreamReader reader = new StreamReader("Problem1.html");

while ((line = reader.ReadLine()) != null)
{
   // Find what we need and save it in the result
}

reader.Close();
```

With this code we will read the input file line by line. Let's think whether we have completed a good first step. **Do you know what we have missed?**

From the code written we will read the input file, but only if it exists. **What if the input file does not exist** or could not be opened for some reason? Our present decision does not deal with these problems. There is another problem in our code too: if an error occurs while reading or processing the data file, it **will not be closed**.

With **File.Exists(…)** we will check if the input file exists. If not – we will display an appropriate message and stop program execution. To avoid the second problem we will use the **try-catch-finally** statement (we may use the **using** statement in C# as well). Thus, if an exception arises, we will process it and will always close the file, which we worked with. We must not forget that the object of the **StreamReader** class must be declared outside the **try** block, otherwise it will be unavailable in the **finally** block. This is not a fatal error, but often made by novice programmers.

It is better to define the **input file name as a constant**, because we probably will use it in several places.

Another thing: when reading from a text file it is appropriate to specify explicitly the **character encoding**. Let's see what we get:

```
using System;
using System.IO;
using System.Text;

class HtmlTagRemover
{
  private const string InputFileName = "Problem1.html";
  private const string Charset = "windows-1251";

  static void Main()
  {
    if (!File.Exists(InputFileName))
    {
      Console.WriteLine(
        "File " + InputFileName + " not found.");
      return;
    }

    StreamReader reader = null;
    try
    {
      Encoding encoding = Encoding.GetEncoding(Charset);
      reader = new StreamReader(InputFileName, encoding);
      string line;
      while ((line = reader.ReadLine()) != null)
      {
```

```
            // Find what we need and save it in the result
        }
      }
      catch (IOException)
      {
        Console.WriteLine(
          "Can not read file " + InputFileName + ".");
      }
      finally
      {
        if (reader != null)
        {
          reader.Close();
        }
      }
    }
}
```

Test the Input File Reading Code

We handled the described problems and it **seems we have implemented the reading of the input file**. We wrote a lot of code. To be convinced that it is correct, we can **test our unfinished code.** For example let's print the content of the input file to the console, and then try processing nonexistent files. The writing will be done in a **while** loop using **Console.WriteLine(…)**:

```
…
while ((line = reader.ReadLine()) != null)
{
  Console.WriteLine(line);
}
…
```

If we test the piece of code we have with the **Problem1.html** sample file from the problem description, the result is correct – the input file itself:

```
<html>
<head>
<title>Welcome to our site!</title>
</head>
<body>
<center>
<img src="/en/img/logo.gif" width="130" height="70" alt="Logo">
<br><br><br>
<font size="-1"><a href="/index.html">Home</a> -
```

```
<a href="/contenst.html">Contacts</a> -
<a href="/about.html">About</a></font><p>
</center>
</body>
</html>
```

Let's try a **nonexistent file**. We change the file name **Problem1.html** with **Problem2.html**. The result is the following:

```
File Problem2.html not found
```

We are convinced that **the code till now is correct**. Let's move to the next step of the implementation of our idea (algorithm).

Step 2 – Remove the Tags

Now we want to find a suitable way to **remove all tags**. How should we do this?

One possible way is to **check the line character by character**. For each character in the current row we will look for the character "**<**". On the right side of it we will know that we have a tag (opening or closing). The end tag character is "**>**". So we can find tags and remove them. To not get the words connected between adjacent tags, each tag will be replaced with the character for a blank line "**\n**".

The algorithm is simple to implement, but isn't there a more clever way? Can we use **regular expressions**? They can easily look for tags and replace them with "**\n**", right? In the same time the code will be simple and in case of errors, they will be removed more easily. We will consider this option. What should we do? First we need to write a regular expression. Here is how it may look:

```
<[^>]*>
```

The idea is simple: any string, that starts with "**<**", continues with arbitrary sequence of characters, other than "**>**" and ends with "**>**" is an HTML tag. Here's how we can **replace the tags with a new line**:

```
private static string RemoveAllTags(string str)
{
   string textWithoutTags = Regex.Replace(str, "<[^>]*>", "\n");
   return textWithoutTags;
}
```

After coding this step, we should test it. For this purpose again we print to the console the strings we found via **Console.WriteLine(…)**. And test the code:

HtmlTagRemover.cs

```
using System;
using System.IO;
using System.Text;
using System.Text.RegularExpressions;

class HtmlTagRemover
{
   private const string InputFileName = "Problem1.html";
   private const string Charset = "windows-1251";

   static void Main()
   {
      if (!File.Exists(InputFileName))
      {
         Console.WriteLine(
            "File " + InputFileName + " not found.");
         return;
      }

      StreamReader reader = null;
      try
      {
         Encoding encoding = Encoding.GetEncoding(Charset);
         reader = new StreamReader(InputFileName, encoding);
         string line;
         while ((line = reader.ReadLine()) != null)
         {
            line = RemoveAllTags(line);
            Console.WriteLine(line);
         }
      }
      catch (IOException)
      {
         Console.WriteLine(
            "Can not read file " + InputFileName + ".");
      }
      finally
      {
         if (reader != null)
         {
            reader.Close();
         }
```

```
    }
  }

  private static string RemoveAllTags(string str)
  {
    string strWithoutTags =
      Regex.Replace(str, "<[^>]*>", "\n");
    return strWithoutTags;
  }
}
```

Testing the Tag Removal Code

Let's **test the program** with the following input file:

```
<html><body>
Click<a href="info.html">on this
link</a>for more info.<br />
This is<b>bold</b>text.
</body></html>
```

The result is as follows:

```
(empty rows)
Click
on this
link
for more info.
(empty row)
This is
bold
text.
(empty rows)
```

Everything **works perfectly**, only that we have **extra blank lines**. Can we remove them? This will be our next step.

Step 3 – Remove the Empty Lines

We can remove unnecessary blank lines, replacing a double blank line "**\n\n**" with a single blank line "**\n**". We should not have groups of more than one character for a new line "**\n**". Here is an example how we can perform the substitution:

```
private static string RemoveDoubleNewLines(string str)
```

```
{
   return str.Replace("\n\n", "\n");
}
```

Testing the Empty Lines Removal Code

As always, before we move forward, we test whether the method works correctly. We try a text, which has no blank rows, and then add 2, 3, 4 and 5 blank lines, including at the beginning and at the end of text.

We find that **the above method does not work correctly**, when there are 4 blank lines one after another. For example, if we submit as input "**ab\n\n\n\ncd**", we will get "**ab\n\n\cd**" instead of "**ab\ncd**". This defect occurs because the **Replace(…)** finds and replaces a single match, scanning the text from left to right. If in result of a substitution the searched string reappears, it is skipped.

See how useful it is when each method is tested on time. We do not end up wondering why the program does not work when we have 200 lines of code, full of errors. **Early detection of defects is very useful** and we should do it whenever possible. Here is the corrected code:

```
private static string RemoveDoubleNewLines(string str)
{
   string pattern = "[\n]+";
   return Regex.Replace(str, pattern, "\n");
}
```

The above code uses a regular expression to find any sequence of **\n** characters and replaces it with a single **\n**.

After a series of tests, we are **convinced that the method works correctly**. We are ready to test the program that removes all unnecessary newlines. For this purpose we make the following changes:

```
while ((line = reader.ReadLine()) != null)
{
   line = RemoveAllTags(line);
   line = RemoveDoubleNewLines(line);
   Console.WriteLine(line);
}
```

We test the code again. Still it **seems there are blank lines**. Where do they come from? Perhaps, if we have a line that contains only tags, it will cause a problem. Therefore we may prevent this case. We add the following checks:

```
if (!string.IsNullOrEmpty(line))
```

```
{
   Console.WriteLine(line);
}
```

This removes most of the blank lines, but not all.

Remove the Empty Lines: Second Attempt

If we think more, it could happen so, that **a line begins or ends with a tag**. Then this tag will be replaced with a single blank line and so at the beginning or at the end of the line we may get a blank line. This means that we should clean the empty rows at the beginning and at the end of each line. Here's how we can make the cleaning:

```
private static string TrimNewLines(string str)
{
   int start = 0;
   while (start < str.Length && str[start] == '\n')
   {
      start++;
   }

   int end = str.Length - 1;
   while (end >= 0 && str[end] == '\n')
   {
      end--;
   }

   if (start > end)
   {
      return string.Empty;
   }

   string trimmed = str.Substring(start, end - start + 1);
   return trimmed;
}
```

The method works very simply: goes from left to right and **skips all newline characters**. Then passes from right to left and skips again all newline characters. If the left and right positions have passed each other, this means that the string is either empty or contains only newlines. Then the method returns an empty string. Otherwise it returns back everything to the right of the start position and to the left of the end position.

Remove the Empty Lines: Test Again

As always, we **test whether the above method works correctly** with several examples, including an empty string, no string breaks, string breaks left or right or both sides and a string with new lines. We make sure, that **the method now works correctly**.

Now we have to modify the logic of processing the input file:

```
while ((line = reader.ReadLine()) != null)
{
   line = RemoveAllTags(line);
   line = RemoveDoubleNewLines(line);
   line = TrimNewLines(line);
   if (!string.IsNullOrEmpty(line))
   {
      Console.WriteLine(line);
   }
}
```

Step 4 – Print Results in a File

It remains to **print the results in the output file**. To print the results in the output file we will use the **StreamWriter**. This step is trivial. We must only consider that writing to a file can cause an exception and that's why we need to change the logic for error handling slightly, opening and closing the flow of input and output to the file.

Here is what we finally get as a **complete source code of the program**:

HtmlTagRemover.cs

```
using System;
using System.IO;
using System.Text;
using System.Text.RegularExpressions;

class HtmlTagRemover
{
   private const string InputFileName = "Problem1.html";
   private const string OutputFileName = "Problem1.txt";
   private const string Charset = "windows-1251";

   static void Main()
   {
      if (!File.Exists(InputFileName))
      {
```

```
            Console.WriteLine(
                "File " + InputFileName + " not found.");
            return;
        }

        StreamReader reader = null;
        StreamWriter writer = null;
        try
        {
            Encoding encoding = Encoding.GetEncoding(Charset);
            reader = new StreamReader(InputFileName, encoding);
            writer = new StreamWriter(OutputFileName, false,
                encoding);

            string line;
            while ((line = reader.ReadLine()) != null)
            {
                line = RemoveAllTags(line);
                line = RemoveDoubleNewLines(line);
                line = TrimNewLines(line);
                if (!string.IsNullOrEmpty(line))
                {
                    writer.WriteLine(line);
                }
            }
        }
        catch (IOException)
        {
            Console.WriteLine(
                "Can not read file " + InputFileName + ".");
        }
        finally
        {
            if (reader != null)
            {
                reader.Close();
            }

            if (writer != null)
            {
                writer.Close();
            }
        }
    }
```

```
/// <summary>
/// Replaces every tag with new line
/// </summary>
private static string RemoveAllTags(string str)
{
   string strWithoutTags =
      Regex.Replace(str, "<[^>]*>", "\n");
   return strWithoutTags;
}

/// <summary>
/// Replaces sequence of new lines with only one new line
/// </summary>
private static string RemoveDoubleNewLines(string str)
{
   string pattern = "[\n]+";
   return Regex.Replace(str, pattern, "\n");
}

/// <summary>
/// Removes new lines from start and end of string
/// </summary>
private static string TrimNewLines(string str)
{
   int start = 0;
   while (start < str.Length && str[start] == '\n')
   {
      start++;
   }

   int end = str.Length - 1;
   while (end >= 0 && str[end] == '\n')
   {
      end--;
   }

   if (start > end)
   {
      return string.Empty;
   }

   string trimmed = str.Substring(start, end - start + 1);
   return trimmed;
```

```
    }
}
```

Testing the Solution

Until now, we were testing the individual steps for the solution of the task. Through the tests of individual steps we reduced the possibility of errors, but that does not mean that we should not test the whole solution. We may have missed something, right? Now let's **thoroughly test the code**.

- Test with the **sample input** file from the problem statement. Everything works correctly.
- Test our **"complex" example**. Everything works fine.
- Test the **border cases** and run an output test.
- We test with a **blank file**. Output is correct – an empty file.
- Test with a file that contains **only one word** "**`Hello`**" and does not contain tags. The result is correct – the output contains only the word "**`Hello`**".
- Test with a **file that contains only tags and no** text. The result is again correct – an empty file.
- Try to **put blank lines** of at the most amazing places in the input file. These empty lines should all be removed. For example we can run the following test:

```
  Hello

<br><br>


<b>I<b> am here

I am not <b>here</b>
```

The result is as follows:

```
  Hello
I
 am here
I am not
Here
```

It seems we found a **small defect**. There is a **space at the beginning of some of the lines**.

Fixing the Leading Spaces Defect

Under the problem description it is not clear whether this is a defect but let's try to **fix it**. We could add the following code when processing the next line of the input file:

```
line = line.Trim();
```

The defect is fixed, but only from the first line. We run the debugger and we notice why it is so. The reason is that we print into the output file a string of characters with value "**I\n am here**" and so we get a space after a blank line. We can correct the defect, by replacing all blank lines, followed by white space (blank lines, spaces, tabs, etc.) with a single blank line. Here is the **correction**:

```
private static string RemoveDoubleNewLines(string str)
{
   string pattern = "\n\\s+";
   return Regex.Replace(str, pattern, "\n");
}
```

We fixed that error too. Now we have only to change this name to a more appropriate one, for example **RemoveNewLinesWithWhiteSpace(…)**.

Now we need to **test again after the "fixes"** in the code (**regression test**). We put new lines and spaces scattered randomly and make sure that everything works correctly now.

Performance Test

One last test remains: **performance**. We can create easily create a large input file. We open a site, for example http://www.microsoft.com, grab the source code and copy it 1000 times. We get a large enough input file. In our case, we get a **44 MB file** with 947,000 lines. Processing it takes under 10 seconds, which is a **perfectly acceptable speed**. When we test the solution we should not forget that the processing of the file depends on our hardware (our test was performed in 2009 on an average fast laptop).

Taking a look at the result, however, we notice a very troublesome problem. There are parts of a tag. More precisely, we see the following:

```
<!--
var s_pageName="home page"
//-->
```

It quickly becomes clear that we **missed a very interesting case**. In an HTML tag can be closed few lines after its opening, e.g. a **single tag may span several consecutive lines**. That was exactly our case: we have a

comment tag that contains JavaScript code. If the program worked correctly, it would have cut the entire tag rather than keep it in the source file.

Did you see how testing is useful and **how testing is important**? In some big companies (like Microsoft) having a solution without tests is considered as only 50% of the work. This means that if you write code for 2 hours, you should spend on testing (manual or automated) at least 2 more hours! This is the only way to create high-quality software.

What a pity that we discovered the problem just now, instead of at the beginning, when we were checking whether our idea for the task is correct, before we wrote the program. Sometimes it happens, unfortunately.

How to Fix the Problem with the Tag at Two Lines?

The first idea that occurs to us is to **load in memory the entire input file** and process it as one big string rather than row by row. This is an idea that seems to work but will run slow and **consume large amounts of memory**. Let's look for another idea.

A New Idea: Processing the Text Char by Char

Obviously **we cannot read the file line by line**. Can we **read it character by character**? If yes, how we will treat tags? It occurs to us that if we read the file character by character, we can know at any moment, whether we are in or outside of a tag, and if we are outside the tag, we can print everything that we read (followed by a new line). We need to avoid adding new lines, as well as and trailing whitespace. We will get something like this:

```
bool inTag = false;
while (! <end of file is reached>)
{
   char ch = (read the next character);
   if (ch == '<')
   {
      inTag = true;
   }
   else if (ch == '>')
   {
      inTag = false;
   }
   else
   {
      if (!inTag)
      {
         PrintBuffer(ch);
      }
   }
```

```
}
```

Implementing the New Idea

The idea is **very simple and easy to implement**. If we implement it directly, we will have a problem with empty lines and the problem of merging text from adjacent tags. To solve this problem, we can accumulate the text in the **StringBuilder** and print it at the end of file or when switching from text to a tag. We will get something like this:

```
bool inTag = false;
StringBuilder buffer = new StringBuilder();
while (! <end of file is reached>)
{
   char ch = (read the next character);
   if (ch == '<')
   {
      if (!inTag)
      {
         PrintBuffer(buffer);
      }
      buffer.Clear();
      inTag = true;
   }
   else if (ch == '>')
   {
      inTag = false;
   }
   else
   {
      if (!inTag)
      {
         buffer.Append(ch);
      }
   }
}
PrintBuffer(buffer);
```

The missing **PrintBuffer(…)** method should clean the whitespace from the text in the buffer and print it in the output followed by a new line. Exception is when we have whitespace only in the buffer (it should not be printed).

We already have most of the code, so **step-by-step implementation mat not be necessary**. We can just replace the pieces of wrong old code with the new code implementing the new idea. If we add the logic for avoiding empty

lines as well as reading input and writing the result we obtain is a **complete solution to the task with the new algorithm**:

SimpleHtmlTagRemover.cs

```
using System;
using System.IO;
using System.Text;
using System.Text.RegularExpressions;

public class SimpleHtmlTagRemover
{
  private const string InputFileName = "Problem1.html";
  private const string OutputFileName = "Problem1.txt";
  private const string Charset = "windows-1251";
  private static Regex regexWhitespace = new Regex("\n\\s+");

  static void Main()
  {
    if (!File.Exists(InputFileName))
    {
      Console.WriteLine(
        "File " + InputFileName + " not found.");
      return;
    }

    StreamReader reader = null;
    StreamWriter writer = null;
    try
    {
      Encoding encoding = Encoding.GetEncoding(Charset);
      reader = new StreamReader(InputFileName, encoding);
      writer = new StreamWriter(OutputFileName, false,
        encoding);
      RemoveHtmlTags(reader, writer);
    }
    catch (IOException)
    {
      Console.WriteLine(
        "Cannot read file " + InputFileName + ".");
    }
    finally
    {
      if (reader != null)
      {
```

```
            reader.Close();
        }
        if (writer != null)
        {
            writer.Close();
        }
    }
}

/// <summary>Removes the tags from a HTML text</summary>
/// <param name="reader">Input text</param>
/// <param name="writer">Output text (result)</param>
private static void RemoveHtmlTags(
    StreamReader reader, StreamWriter writer)
{
    StringBuilder buffer = new StringBuilder();
    bool inTag = false;
    while (true)
    {
        int nextChar = reader.Read();
        if (nextChar == -1)
        {
            // End of file reached
            PrintBuffer(writer, buffer);
            break;
        }
        char ch = (char)nextChar;
        if (ch == '<')
        {
            if (!inTag)
            {
                PrintBuffer(writer, buffer);
            }
            buffer.Clear();
            inTag = true;
        }
        else if (ch == '>')
        {
            inTag = false;
        }
        else
        {
            // We have other character (not "<" or ">")
            if (!inTag)
```

```
            {
               buffer.Append(ch);
            }
         }
      }
   }

   /// <summary>Removes the whitespace and prints the buffer
   /// in a file</summary>
   /// <param name="writer">the result file</param>
   /// <param name="buffer">the input for processing</param>
   private static void PrintBuffer(
      StreamWriter writer, StringBuilder buffer)
   {
      string str = buffer.ToString();
      string trimmed = str.Trim();
      string textOnly = regexWhitespace.Replace(trimmed, "\n");
      if (!string.IsNullOrEmpty(textOnly))
      {
         writer.WriteLine(textOnly);
      }
   }
}
```

The input file is **read character by character** with the class **StreamReader**.

Originally the buffer for accumulating of text is empty. In the main loop we **analyze each read character**. We have the following cases:

- If we get to the **end of file**, we print whatever is in the buffer and the algorithm ends.
- When we encounter the character "**<**" (**start tag**) we first print the buffer (if we find that the transition is from text to tag). Then we clear the buffer and set **inTag** = **true**.
- When we encounter the character "**>**" (**end tag**) we set **inTag** = **false**. This will allow the next characters after the tag to accumulate in the buffer.
- When we encounter **another character** (text or blank space), it is added to the buffer, if we are outside tags. If we are in a tag the character is ignored.

Printing of the buffer takes care of **removing empty lines** in text and clearing the empty space at the beginning and end of text (trimming the leading and trailing whitespace). How exactly we do this, we already discussed in the previous solution of the problem.

In the second solution the processing of the buffer is much lighter and shorter, so the buffer is processed immediately before printing.

In the previous solution of the task we used regular expressions for replacing with the static methods of the class **Regex**. For **improved performance** now we create the regular expression object just once (as a **static** field). Thus the regular expression pattern is compiled just once to a state machine.

Testing the New Solution

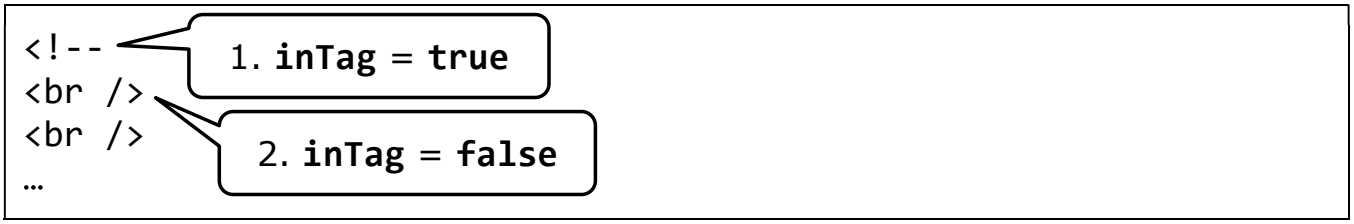
It remains to **test thoroughly the new solution**. We have to perform all tests conducted on the previous solution. Add test with tags, which are spread over several lines. Again, test performance with the Microsoft website copied 1000 times. Assure that the program works correctly and is even faster.

Let's try with another site, such as the official website of this book – http://www.introprogramming.info (as of April 2011). Again, take the source code of the site and run the solution of our task with it. After carefully reviewing the input data (source code on the website of the book) and the output file, we notice that **there is a problem again**. Some content of this tag is printed in the output file:

```
<!--
<br />
<br />
Read the free book by Svetlin Nakov and team for developing with
Java.
…
…
-->
```

Where Is the Problem?

The problem seems to occur when **one tag meets another tag, before the first tag is closed**. This can happen in HTML comments. Here's how to get to the error:



As we know, in the solution of the task we use Boolean variable (**inTag**), to know whether the current character is in the tag or not. On the figure above we have shown that in moment 1 we set **inTag** = **true**. So far so good. Then comes moment 2, where the current character read is "**>**". At this point we find **inTag** = **false**. The problem is that the tag, which is open from moment 1 is not yet closed, and the Boolean variable indicates that we are not in the

tag anymore and the following characters are saved in the buffer. If between the two tags for a new line (**
**) we have text, it would also be saved in the buffer.

How to Fix the Problem?

It turned out that in the second solution **there is a mistake**. The program does not work correctly in the presence of **nested tags in a comment tag**. By Boolean variable can only know whether we are in a tag or not, but cannot remember if we are still in the preceding. This tells us that instead of using a Boolean variable, we can store the number of tags in which we are (in variable of type **int – tag counter**). We will modify the solution:

```
int openedTags = 0;
StringBuilder buffer = new StringBuilder();
while (! <end of file is reached>)
{
   char ch = (read the next character);
   if (ch == '<')
   {
      if (openedTags == 0)
      {
         PrintBuffer(buffer);
      }
      buffer.Remove(0, buffer.Length);
      openedTags++;
   }
   else if (ch == '>')
   {
      openedTags--;
   }
   else
   {
      if (openedTags == 0)
      {
         buffer.Append(ch);
      }
   }
}
PrintBuffer(buffer);
```

In the main loop we analyze each read character. We have the following cases:

- If we get to the **end of the file**, print whatever is in the buffer and the **algorithm ends**.

- When we encounter the character "**<**" (**start tag**) first we print the buffer (if we find that the transition from text to the tag). Then we clear the buffer and **increase the counter** by one.
- When we encounter the character "**>**" (**end tag**) we **reduce the counter** by one. Closing of a nested tag will not allow accumulation in the buffer. If after closing a tag we are out of all tags, the characters will begin to accumulate in the buffer.
- When we encounter **another character** (text or blank space), it is **added to the buffer**, if we are outside all tags. If we are inside a tag – the **character is ignored**.

It remains to **write the whole solution again and then test it**. The logic for reading the input file and printing the buffer remains the same:

SimpleHtmlTagRemover.cs

```
using System;
using System.IO;
using System.Text;
using System.Text.RegularExpressions;

public class SimpleHtmlTagRemover
{
  private const string InputFileName = "Problem1.html";
  private const string OutputFileName = "Problem1.txt";
  private const string Charset = "windows-1251";
  private static Regex regexWhitespace = new Regex("\n\\s+");

  static void Main()
  {
    if (!File.Exists(InputFileName))
    {
      Console.WriteLine(
        "File " + InputFileName + " not found.");
      return;
    }

    StreamReader reader = null;
    StreamWriter writer = null;
    try
    {
      Encoding encoding = Encoding.GetEncoding(Charset);
      reader = new StreamReader(InputFileName, encoding);
      writer = new StreamWriter(OutputFileName, false,
        encoding);
```

```
            RemoveHtmlTags(reader, writer);
        }
        catch (IOException)
        {
            Console.WriteLine(
                "Cannot read file " + InputFileName + ".");
        }
        finally
        {
            if (reader != null)
            {
                reader.Close();
            }
            if (writer != null)
            {
                writer.Close();
            }
        }
    }

    /// <summary>Removes the tags from a HTML text</summary>
    /// <param name="reader">Input text</param>
    /// <param name="writer">Output text (result)</param>
    private static void RemoveHtmlTags(
        StreamReader reader, StreamWriter writer)
    {
        int openedTags = 0;
        StringBuilder buffer = new StringBuilder();
        while (true)
        {
            int nextChar = reader.Read();
            if (nextChar == -1)
            {
                // End of file reached
                PrintBuffer(writer, buffer);
                break;
            }
            char ch = (char)nextChar;
            if (ch == '<')
            {
                if (openedTags == 0)
                {
                    PrintBuffer(writer, buffer);
                    buffer.Length = 0;
```

```
            }
            openedTags++;
        }
        else if (ch == '>')
        {
            openedTags--;
        }
        else
        {
            // We aren't in tags (not "<" or ">")
            if (openedTags == 0)
            {
                buffer.Append(ch);
            }
        }
    }
}

/// <summary>Removes the whitespace and prints the buffer
/// in a file</summary>
/// <param name="writer">the result file</param>
/// <param name="buffer">the input for processing</param>
private static void PrintBuffer(
    StreamWriter writer, StringBuilder buffer)
{
    string str = buffer.ToString();
    string trimmed = str.Trim();
    string textOnly = regexWhitespace.Replace(trimmed, "\n");
    if (!string.IsNullOrEmpty(textOnly))
    {
        writer.WriteLine(textOnly);
    }
}
}
```

Testing the New Solution

Again we **test the solution of the problem**. We **perform all tests** made on the previous solution (see section "Testing the Solution"). We also try the site of MSDN (http://msdn.microsoft.com). Let's carefully check the output file. We can see that at its end the file contains wrong characters (in April 2011). After carefully reviewing the source code of the MSDN site, we notice that there is an incorrect representation of the character "**>**" (to visualize this character in the HTML document "**>**" should be used, not "**>**"). However, this is an error in the MSDN site, not in our program.

Now it remains to **test the performance** of our program with the site of this book (http://www.introprogramming.info) copied 1000 times. We assure that the program works fast enough for it too.

Finally we are **ready for the next task**.

Problem 2: Escape from Labyrinth

We are given a **labyrinth**, which consists of **N x N squares** and each of it can be **passable (0) or not (x)**. Our hero Jack is in one of the squares (*****):

x	x	x	x	x	x
0	x	0	0	0	x
x	*	0	x	0	x
x	x	x	x	0	x
0	0	0	0	0	x
0	x	x	x	0	x

Two of the squares are **neighboring**, if they have a common wall. In one step Jack can pass from one passable square to its neighboring passable square. If Jack steps in a cell, which is on the edge of the labyrinth, he can go out from the labyrinth with one step.

Write a program, which by a given labyrinth prints the **minimal number of steps**, which Jack needs, **to go out from the labyrinth** or **-1** if there is no way out.

The input data is read from a text file named **`Problem2.in`**. On the first line of the file is the number N (2 < N < 100). On the each of next N lines there are N characters, each of them is either "0" or "x" or "*". The output is one number and must be in the file **`Problem2.out`**.

Sample input – **`Problem2.in`**:

```
6
xxxxxx
0x000x
x*0x0x
xxxx0x
00000x
0xxx0x
```

Sample output – **`Problem2.out`**:

```
9
```