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

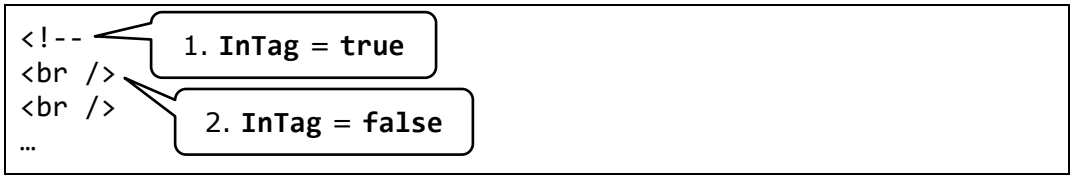
It remains to **test thoroughly the new solution**. We have to perform all tests conducted on the previous solution. Add test with tags, which are spread over several lines. Again, test performance with the Microsoft website copied 1000 times. Assure that the program works correctly and is even faster.

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
```

Figure Out an Idea for a Solution

We have a labyrinth and we should **find the shortest path** in it. This is not an easy task and we should think a lot or we should read somewhere how to solve such kinds of tasks.

Our algorithm will begin its movement from the initial point we are given. We know we can move to a neighboring cell horizontally or vertically, but not diagonally. Our algorithm must traverse the labyrinth in some way, to find the shortest path in it. How to traverse the cells in the labyrinth?

One possible decision is the following: we start from the initial cell. Move to one of its neighboring cells, after this in a neighboring cell of the current (which is passable and still unvisited), after this in a neighboring cell of the last visited (which is passable and still unvisited) and we go on forward recursively until we reach an exit of the labyrinth, or we reach a place where we can't continue (there is no neighboring cell which is free or unvisited). In this moment we go back from the recursion (to the previous cell) and visit another neighboring cell for the previous cell. If we can't continue, we go back again. The described **recursive process** is the process of traversing the labyrinth **in depth** (remember the chapter "Recursion" and DFS traversal).

The question "Is it needed to walk through one cell more than once" occurs to us? If we walk through one cell at most once, we can walk through the whole labyrinth faster and if there is an exit, we will find it. But **will this be the minimal path?** If we draw the whole process on a paper, we will find out quickly the path will not be the minimal.

If we mark the cell we leave on the way back of the recursion as free, this will allow us to reach each cell repeatedly, coming from a different path. The **full recursive walk of the labyrinth will find all possible paths** from the initial cell to any other cell. From all the found paths we can **choose the shortest path** to a cell on the bound of the labyrinth (exit) and that's how we will find a solution for the problem.

Verification of the Idea

It seems we have an idea for solving the problem: with recursive walk we **find all the possible paths** in the labyrinth from the initial cell to a cell on the bounds of the labyrinth and from all these paths we **choose the shortest one**. Let's check the idea.

We take a sheet of paper and make one example of the labyrinth. We try the algorithm. It's obvious it finds all the paths from the initial cell to the one of the exits and it travels a lot forwards-backwards. As a result it finds all exits and among all paths it can be chosen the shortest one.

Does the idea work **if there is no exit**? We create a second labyrinth, which is without exit. We try out the algorithm on it, again on a sheet of paper. We see after long circulation forwards-backwards that the algorithm does not find an exit and finishes.

It looks we have a correct idea for solving the problem. Let's move forward and think for the data structures.

What Data Structures to Use?

First, we have to decide **how to store the labyrinth**. It's natural to use a matrix of characters, just as the one on the figure. We will consider that one cell is passable and we can enter it, if it has a character, different from the character '**x**'. We can store the labyrinth in a matrix of numbers and Boolean values, but the difference is not significant. The matrix of characters is comfortable for printing, and this will help us while debugging. There are not many options. We will store the labyrinth in a **matrix of characters**.

After this, we have to decide in what structure to **keep the visited** through the recursion (current path) **cells**. We always need the last visited cell. This leads us to a structure, which is "last in, first out", i.e. **stack**. We can use `Stack<Cell>`, where `Cell` is a class, containing the coordinates of one cell (number of row and number of column). It remains to think where to keep the found paths, to find the shortest of them. If we think of it, it is not necessary to keep all the paths. It is enough to keep the current path and the shortest till this moment. It's not even necessary to keep the shortest path till this moment but only its **length**. Every time we find a path to an exit of the labyrinth we can take its length and if it is shorter than the shortest path to this moment to keep it.

It seems we found **efficient data structures**. According to our recommendations for problem solving, it is early to write the code of the program, because we have to think of the efficiency of the algorithm.

Think About the Efficiency

Let's check our idea against **efficiency**. What are we doing? We find all the possible paths and we take the shortest. There's no argument the algorithm will not work, but if the labyrinth is way bigger, will it work fast?

To answer this question, we should think **how much paths there are**. If we take an empty labyrinth, on the each step of the recursion we will have an average number of 3 free cells to go (without the cell we are coming from).

If we have for example a labyrinth 10x10, the path could be 100 cells and while we travel on each step we will have 3 neighboring cells. It seems the numbers of paths are sort of 3 to the power of 100. It's obvious **the algorithm will slow down the computer very much and very fast**.

We found a serious problem with the algorithm. It **will work very slowly**, even with small labyrinths, and with bigger ones it will not work at all! The good news is that we haven't written a single line of code and the general change of our approach to the problem will not cost us much time.

Think of Another Idea

We found that **walking through all the paths in the labyrinth is wrong approach**, so we have to think of another.

Let's start with the initial cell and walk through all its neighboring cells and mark them as visited. **For each visited cell we can keep a number equal to the number of cells, which we have travelled to reach it** (the length of the minimal path from the initial cell to the current cell).

For the initial cell the length of the path is 0. For its neighboring cells it should be 1, because we can reach them from the initial cell with one move. For the neighboring cells for the neighbors of the initial cell the length of the path is 2. We can continue this way and we will get to **the following algorithm**:

1. Write the length of the path 0 for the initial cell. Mark it as visited.
2. For each neighboring cell to the initial we mark the length of the path is 1. Mark these cells as visited.
3. For each cell, which is, neighboring to a cell with length of the path 1 and it is not visited, write the length of the path is 2. Mark the cells as visited.
4. Continuing analogous, on the N step we find all the still unvisited cells, which are on a distance of N moves from the initial cell and mark them as visited.

Check the New Idea

To check whether the new idea for solving the "Escape from the Labyrinth" problem is correct we can **visualize the process.** We take another labyrinth to test our idea in a better way. At each step **k** our goal is to fill with the number **k** all cells that can be reached in **k** steps. If at step 0 we fill the initial cell with 0, at step 1 we fill all cells reachable in 1 step from the initial cell, at step 2 we fill all cells reachable in 2 steps, etc. we will be sure that when we fill a cell with a number, this number reflects **the minimal number of steps to reach this cell** starting from the initial cell, right?

Step 0 – mark the distance from the initial cell to itself with **0** (mark the free cells with "**-**"):

x	x	x	x	x	x
-	x	-	-	-	x
x	0	-	x	-	x
x	-	-	x	-	x
x	-	-	-	-	x
-	x	x	x	-	x

Step 1– mark with **1** all the neighbors to cells with a value of 0:

x	x	x	x	x	x
-	x	-	-	-	x
x	0	1	x	-	x
x	1	-	x	-	x
x	-	-	-	-	x
-	x	x	x	-	x

Step 2 – mark with **2** all the passable neighbors to cells with value 1:

x	x	x	x	x	x
-	x	2	-	-	x
x	0	1	x	-	x
x	1	2	x	-	x
x	2	-	-	-	x
-	x	x	x	-	x

Step 3 – mark with 3 all passable neighbors to cells with value of 2:

x	x	x	x	x	x
-	x	2	3	-	x
x	0	1	x	-	x
x	1	2	x	-	x
x	2	3	-	-	x
-	x	x	x	-	x

Continuing this way, in a moment either we will reach a cell at the edge of the labyrinth (an exit) or we will find such a cell is unreachable. It seems like **our algorithm works correctly**. It will either find an exit or will find that there is no reachable exit. If at some step an exit is found, the path to it will be guaranteed to be **the shortest possible** (otherwise the exit should already be found at some of the earlier steps).

Breaking the Problem into Subproblems

Having invented the idea for solving the labyrinth escaping problem, it will be easy to break it into subproblems. The main subproblems could be: **reading the input** labyrinth, **finding the shortest path** to some of its exits and **printing the results**. The **path finding subproblem** could be further divided into subproblems (steps) which we discussed in the previous section.

Checking the Performance of the New Algorithm

Because we never visit a cell more than once, the number of steps, which this algorithm does, **should not be big**. For example, if we have a labyrinth with size 100 x 100, it will have 10,000 cells, we will visit each of the cells at most once and for each of them we will check every neighbor if it is free, i.e. we will check 4 times each cell. At the end we will do at most 40,000 checks and we will visit at most 10,000 cells. We will do a total amount of 50,000 operations. This means **the algorithm will work instantly**.

Check If the New Algorithm Is Correct

It seems this time we don't have a problem with the performance. We have a **fast algorithm**.

Let's check if it is correct. For this purpose we draw a bigger and more complex example on a sheet of paper, which has many exits and a lot of paths, and we begin to perform the algorithm. After this we try with a labyrinth with no exit. It seems the algorithm ends, but does not find an exit so it's working. We try another 2-3 examples and **convince ourselves this algorithm always finds the shortest path to an exit and always works fast**, because it visits each of the cells of the labyrinth at most once.

What Data Structures to Use?

With the new algorithm we walk consequently through all neighboring cells to the initial cell. We can put them into a **data structure**, for example in an array or better a list (or list of lists), because we can't add in the array.

Then we take the **list of the reached cells on the last step** and we add their neighbors in another list.

That's how if we index the lists we have **list_0**, which contains **the initial cell**, **list_1**, which contains passable **neighboring cells to the initial**, after this **list_2**, which contains **passable neighbors to list_1** and so on. At the N step we have the **list_n**, which contains all the cells, which we can **reach in exactly N steps**, i.e. which are **at a distance of n** from the initial cell.

It seems we can use a list of lists, to keep the cells on each step. If we think about it, to get the n list, we need the (**n-1**)-list. So it seems we don't need list of lists but **only the list from the last step**.

We can make general conclusion: cells are processed in the order of entry: when the cells of step **k** are finished, then we process the cells from step **k+1**, and just after them – the cells from step **k+2**, and so on. The process seems like a **queue**: earlier accessed cells are processed earlier. If we dig a bit inside, we will conclude, that we have just **re-invented the Breadth-First-Search algorithm** (read about BFS in Wikipedia).

To implement the BFS algorithm we can use a **queue of cells**. For this purpose we have to **define class `Cell`**, which contains the coordinates of

given cell (row and column). We can **keep the distance** from each cell to the initial cell in a matrix. If the distance is not calculated yet, we store -1.

If we think a little more, the distance from the initial cell can be kept in the cell itself (in the class **Cell**) instead of creating a special matrix for the distances. That way we will save memory.

Now we are clear about the data structures. Now we have to implement the algorithm step by step.

Step 1 – The Class Cell

We can begin with the **definition of the Cell class**. We need it to save the initial cell, from which begins the searching of the path. We will use auto-implemented properties to make the code shorter and more readable. Here is the **Cell** class:

```
public class Cell
{
   public int Row { get; set; }
   public int Column { get; set; }
   public int Distance { get; set; }
}
```

We can add a **constructor** to simplify the way we use this class:

```
public Cell(int row, int column, int distance)
{
   this.Row = row;
   this.Column = column;
   this.Distance = distance;
}
```

Generally it is a good idea **to test the code after each step**, but the above code is too simple to be tested. We will test is later as part of some more complex piece of code.

Step 2 – Reading the Input File

We will read the input file **line by line** using the well-known class **StreamReader**. On the each of the lines we will analyze the characters and we will write them in a matrix of characters. When we reach the character "*****" we will keep its coordinates in an instance of class **Cell** to know where to start the searching of the shortest path for getting out of the labyrinth.

We can **define a class Maze** and keep the matrix of the labyrinth and the initial cell in it:

Maze.cs

```
public class Maze
{
   private char[,] maze;
   private int size;
   private Cell startCell = null;

   public void ReadFromFile(string fileName)
   {
      using (StreamReader reader = new StreamReader(fileName))
      {
         // Read the maze size and create the maze
         this.size = int.Parse(reader.ReadLine());
         this.maze = new char[this.size, this.size];

         // Read the maze cells from the file
         for (int row = 0; row < this.size; row++)
         {
            string line = reader.ReadLine();
            for (int col = 0; col < this.size; col++)
            {
               this.maze[row, col] = line[col];
               if (line[col] == '*')
               {
                  this.startCell = new Cell(row, col, 0);
               }
            }
         }
      }
   }
}
```

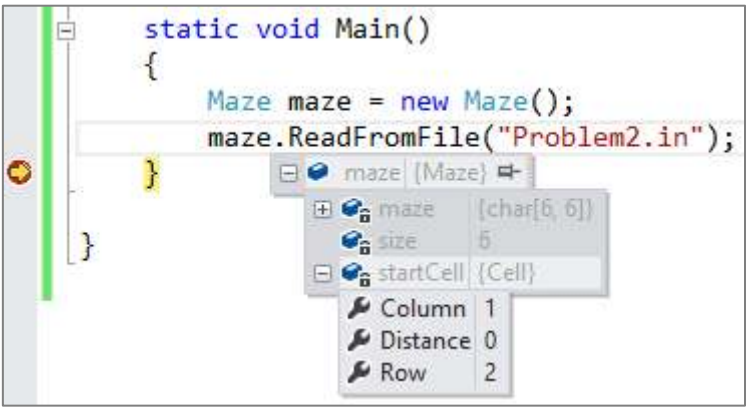
For simplicity we will skip processing the errors while reading and writing in a file. When an **exception occurs** we will skip to catch it in the main method and thus we will leave the CLR to print it on the console.

Testing the Input File Reading Code

We already have the class **Maze** and appropriate representation of data of the input file. To be sure the written so far is correct **we should test**. We can check if the matrix is truly filled as we print it on the console. The other possibility is to view the values of the fields in the class **Maze** through the debugger of Visual Studio. We add a **Main()** method which invokes the maze reading method and we test it:

```
static void Main()
{
  Maze maze = new Maze();
  maze.ReadFromFile("Problem2.in");
}
```

Through the Visual Studio debugger we get convinced that **the input file is correctly read** from the input file:



Step 3 – Finding the Shortest Path

We can **implement the algorithm** directly from what we already discussed. We must define a queue and put in its beginning the initial cell. Afterwards we must take the cell in turn from the queue and add all of its passable unvisited neighbors in a loop. At each step there is a chance to enter in a cell, which is at the border of the labyrinth, and we see we have found an exit and the searching ends. We repeat the loop until the queue is empty. At each visitation of a given cell we check if it is free and if it is, we mark it as impassable. This way we avoid repeatedly visiting the same cell.

Here is how **the implementation of the algorithm** looks like:

```
public int FindShortestPath()
{
  // Queue for traversing the cells in the maze
  Queue<Cell> visitedCells = new Queue<Cell>();
  VisitCell(visitedCells, this.startCell.Row,
    this.startCell.Column, 0);

  // Perform Breath-First-Search (BFS)
  while (visitedCells.Count > 0)
  {
    Cell currentCell = visitedCells.Dequeue();
    int row = currentCell.Row;
```

```
      int column = currentCell.Column;
      int distance = currentCell.Distance;
      if ((row == 0) || (row == size - 1)
        || (column == 0) || (column == size - 1))
      {
        // We are at the maze border
        return distance + 1;
      }
      VisitCell(visitedCells, row, column + 1, distance + 1);
      VisitCell(visitedCells, row, column - 1, distance + 1);
      VisitCell(visitedCells, row + 1, column, distance + 1);
      VisitCell(visitedCells, row - 1, column, distance + 1);
    }

    // We didn't reach any cell at the maze border -> no path
    return -1;
  }

  private void VisitCell(Queue<Cell> visitedCells,
    int row, int column, int distance)
  {
    if (this.maze[row, column] != 'x')
    {
      // The cell is free --> visit it
      maze[row, column] = 'x';
      Cell cell = new Cell(row, column, distance);
      visitedCells.Enqueue(cell);
    }
  }
```

Checking after Step 3

Before the next step, **we must test, to check our algorithm**. We must try the **normal case** and the **border cases**, when there is no exit, when we step on an exit, when the input file doesn't exist or the square matrix is with size of 0. Only then can we start doing the next step. Let's start with **testing the normal (typical) case**. We create the following code to quickly test it:

```
static void Main()
{
  Maze maze = new Maze();
  maze.ReadFromFile("Problem2.in");
  Console.WriteLine(maze.FindShortestPath());
}
```

We run the above code over the sample input file from the problem description and **it works**. The code correctly returns the length of the shortest path to the nearest exit:

```
9
```

Now let's test the **border cases**, e.g. a **labyrinth of size 0**. Unfortunately we get the following result:

```
Unhandled Exception: System.NullReferenceException: Object
reference not set to an instance of an object.
 at Maze.FindShortestPath()
```

We've made a **mistake**. The problem is when the variable, in which we keep the **initial cell**, is initialized with **null**. This can happen in many scenarios. If the labyrinth has no cells (e.g. size of 0) or the initial cell is missing, the result that the program **should return is -1, but not an exception**.

To **fix the bug** we just found we can add a check in the beginning of the **FindShortestPath()** method:

```
public int FindShortestPath()
{
  if (this.startCell == null)
  {
    // Start cell is missing -> no path
    return -1;
  }
  …
```

We **retest the code** with the typical and the border cases. After the fix **it seems the algorithm works correctly** now.

Step 4 – Writing the Result to a File

It remains to write the result of the **FindShortestPath()** to the output file. This is a trivial problem:

```
public void SaveResult(String fileName, int result)
{
  using (StreamWriter writer = new StreamWriter(fileName))
  {
    writer.WriteLine("The shortest way is: " + result);
  }
}
```

Here is how **the complete source code of the solution** looks:

Maze.cs

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

public class Maze
{
   private const string InputFileName = "Problem2.in";
   private const string OutputFileName = "Problem2.out";

   public class Cell
   {
      public int Row { get; set; }
      public int Column { get; set; }
      public int Distance { get; set; }

      public Cell(int row, int column, int distance)
      {
         this.Row = row;
         this.Column = column;
         this.Distance = distance;
      }
   }

   private char[,] maze;
   private int size;
   private Cell startCell = null;

   public void ReadFromFile(string fileName)
   {
      using (StreamReader reader = new StreamReader(fileName))
      {
         // Read maze size and create maze
         this.size = int.Parse(reader.ReadLine());
         this.maze = new char[this.size, this.size];

         // Read the maze cells from the file
         for (int row = 0; row < this.size; row++)
         {
            string line = reader.ReadLine();
            for (int col = 0; col < this.size; col++)
            {
               this.maze[row, col] = line[col];
```

```
            if (line[col] == '*')
            {
              this.startCell = new Cell(row, col, 0);
            }
          }
        }
      }

  public int FindShortestPath()
  {
    if (this.startCell == null)
    {
      // Start cell is missing -> no path
      return -1;
    }

    // Queue for traversing the cells in the maze
    Queue<Cell> visitedCells = new Queue<Cell>();
    VisitCell(visitedCells, this.startCell.Row,
      this.startCell.Column, 0);

    // Perform Breath-First-Search (BFS)
    while (visitedCells.Count > 0)
    {
      Cell currentCell = visitedCells.Dequeue();
      int row = currentCell.Row;
      int column = currentCell.Column;
      int distance = currentCell.Distance;
      if ((row == 0) || (row == size - 1)
        || (column == 0) || (column == size - 1))
      {
        // We are at the maze border
        return distance + 1;
      }

      VisitCell(visitedCells, row, column + 1, distance + 1);
      VisitCell(visitedCells, row, column - 1, distance + 1);
      VisitCell(visitedCells, row + 1, column, distance + 1);
      VisitCell(visitedCells, row - 1, column, distance + 1);
    }

    // We didn't reach any cell at the maze border -> no path
    return -1;
```

```
    }

    private void VisitCell(Queue<Cell> visitedCells,
      int row, int column, int distance)
    {
      if (this.maze[row, column] != 'x')
      {
        // The cell is free --> visit it
        maze[row, column] = 'x';
        Cell cell = new Cell(row, column, distance);
        visitedCells.Enqueue(cell);
      }
    }

    public void SaveResult(string fileName, int result)
    {
      using (StreamWriter writer = new StreamWriter(fileName))
      {
        writer.WriteLine(result);
      }
    }

    static void Main()
    {
      Maze maze = new Maze();
      maze.ReadFromFile(InputFileName);
      int pathLength = maze.FindShortestPath();
      maze.SaveResult(OutputFileName, pathLength);
    }
}
```

Testing the Complete Solution of the Problem

After we have a solution of the problem **we must test it**. We have already tested **the typical case** and **the border cases** (**like missing exit** or **when the initial position stays at the labyrinth edge**). We will execute these tests again to get convinced that the algorithm behaves correctly:

Input	Output	Input	Output	Input	Output	Input	Output
0	-1	2 00 xx	-1	3 0x0 x*x 0x0	-1	3 000 000 00*	1

The algorithm works correctly. The output for each of the test is correct.

It remains to **test with a large labyrinth** (performance test), for example 1000 x 1000. We can make such a labyrinth very easy – with copy / paste. We perform the test and we convince ourselves the program is **working correctly for the big test** and works extremely fast – **there is no delay**.

While testing **we should try every way to break our solution**. We run a few **more difficult examples** (for example a labyrinth with passable cells in the form of **spiral**). We can put large labyrinth with a lot of paths, but without exit. We can try whatever else we wish.

At the end we make sure, that **we have a correct solution** and we pass to the next problem from the exam.

Problem 3: Store for Car Parts

A company is planning to create a system for **managing a store for auto parts**. A single **part** can be used for different car models and it has following characteristics: **code**, **name**, **category** (e.g. suspension, tires and wheels, engine, accessories and etc.), **purchase price**, **sale price**, **list of car models**, with which it is compatible (each car is described with brand, model and year of manufacture, e. g. Mercedes C320, 2008) and **manufacturing company**. **Manufacturing companies** are described with name, country, address, phone and fax.

Design a set of classes with relationships between them, which model the data for the store. Write a **demonstration program**, which demonstrates the classes and their all functionality work correctly with some sample data.

Inventing an Idea for Solution

We have a non-algorithmic problem which is intended to check whether the students at the exam know how to use **object-oriented programming** (OOP), how to design classes and relationships between them to model real-world objects (**object-oriented analysis and design**) and how to use appropriate **data structures** to hold collections of objects.

We are required to create an aggregation of classes and relationships between them, which have to describe the data of the store. We have to find **which nouns are important** for solving the problem. They are objects from the real world, which correspond to **classes**.

Which are these nouns that interest us? We have a **store**, **car parts**, **cars** and **manufacturing companies**. We have to create a class defining a store. It could be named **`Shop`**. Other classes are **`Part`**, **`Car`** and **`Manufacturer`**. In the requirements of the problem there are other nouns too, like code for one part or year of manufacturing of given car. For these nouns we are not creating individual classes, but instead these will be fields in the already created classes. For example in the **`Part`** class there will be let's say a field **`code`** of **`string`** type.

We already know **which will be our classes, and fields to describe them**. We have to identify the relationships between the objects.

Checking the Idea

We **will not check the idea** because there is nothing to be proven with examples and counterexamples or checked whether it will work. We need to write few classes to model a real-world situation: a store for car parts.

What Data Structures to Use to Describe the Relationship between Two Classes?

The data structures, needed for this problem, are of two main groups: **classes** and **relationships between the classes**. The interesting part is how to describe relationships.

To **describe a relationship** (link) between two classes we can use an **array**. With an array we have access of its elements by index, but once it is created we can't change its length. This makes it **uncomfortable for our problem**, because we don't know how many parts we will have in the store and more parts can be delivered or somebody can buy parts so we have to delete or change the data. **`List<T>` is more comfortable**. It has the advantages of an array and also is with variable length and it is easy to add or delete elements.

So far it seems **`List<T>`** is the **most appropriate** for holding aggregations of objects inside another object. To be convinced we will analyze a few more data structures. For example **hash-table** – it is not appropriate in this case, because the structure "parts" is not of the key-value type. It would be appropriate if each of the parts in the store has unique number (e.g. barcode) and we needed to search them by this unique number. Structures like **stack** and **queue** are inappropriate.

The structure "**set**" and its implementation **`HashSet<T>`** is used when we have **uniqueness** for given key. It would be good sometimes to use this structure to avoid duplicates. We must recall that **`HashSet<T>`** requires the methods **`GetHashCode()`** and **`Equals(…)`** to be correctly defined by the **`T`** type.

Our final decision is to use **`List<T>`** for the aggregations and **`HashSet<T>`** for the aggregations which require uniqueness.

Dividing the Task into Subtasks

Now we have to think from where to start writing the code. If we start to write the **`Shop`** class, we will need the **`Part`** class. This reminds us we will have to start with a class, which does not depend on others. We will divide the writing of each class to a **subtask**, and we will start from the independent classes:

- Class describing a car – **`Car`**
- Class describing manufacturer of parts – **`Manufacturer`**
- Class or enumeration for the categories of the parts – **`PartCategory`**

- Class describing part for a car – **Part**
- Class for the store – **Shop**
- Class for testing rest of the classes with sample data – **TestShop**

Implementation: Step by Step

We start writing classes, which we described in our idea. We will create them in the same sequence as in the list above.

Step 1: The Class Car

We start solving the problem by **defining the class Car**. In the definition we have three fields, which keep the manufacturer, the model and the year of manufacturing of the car and the standard method **ToString()**, which returns a human-readable string holding the information about the car. We define the class **Car** in the following way:

Car.cs

```
public class Car
{
   private string brand;
   private string model;
   private int productionYear;

   public Car(string brand, string model, int productionYear)
   {
      this.brand = brand;
      this.model = model;
      this.productionYear = productionYear;
   }

   public override string ToString()
   {
      return "<" + this.brand + "," + this.model + ","
         + this.productionYear + ">";
   }
}
```

Note that the class **Car** is designed to be **immutable**. This means that once created, the car's properties cannot be later modified. This design is not always the best choice. Sometimes we want the class properties to be freely modifiable; sometimes. For our case the immutable design will work well.

Testing the Class Car

Once we have the class **Car**, we could test it by the following code:

```
Car bmw316i = new Car("BMW", "316i", 1994);
Console.WriteLine(bmw316i);
```

The result is as expected:

```
<BMW,316i,1994>
```

We are convinced the class **Car** is correct so far and we can continue with the other classes.

Step 2: The Class Manufacturer

We have to implement the **definition of the class** `Manufacturer`, which describes the manufacturer for given part. It will have five fields – name, country, address, phone number and fax. The class will be **immutable**, because we will not need to change its members after creation. We also define the standard method **`ToString()`** for representing the object as human-readable string.

Manufacturer.cs

```
public class Manufacturer
{
   private string name;
   private string country;
   private string address;
   private string phoneNumber;
   private string fax;

   public Manufacturer(string name, string country,
      string address, string phoneNumber, string fax)
   {
      this.name = name;
      this.country = country;
      this.address = address;
      this.phoneNumber = phoneNumber;
      this.fax = fax;
   }

   public override string ToString()
   {
      return this.name + " <" + this.country + "," + this.address
         + "," + this.phoneNumber + "," + this.fax + ">";
   }
}
```

Testing the Class Manufacturer

We test the class **Manufacturer** just like we tested the class **Car**. It works.

Step 3: The Part Category Enumeration

Part categories are **fixes set of values** and do not have additional details (like name, code and description). This makes them perfect to be modeled as **enumeration**:

PartCategory.cs

```
public enum PartCategory
{
   Engine,
   Tires,
   Exhaust,
   Suspention,
   Brakes
}
```

Step 4: The Class Part

Now we have to **define the class Part**. Its definition will include the following fields: name, code, category, list with cars, where we can use the given part, starting and closing price and manufacturer. Here we will use the data structure **HashSet<Car>** to hold all compatible cars.

The field that keeps the manufacturer of the part will be of **Manufacturer** class, because the task requires us to keep additional information about the manufacturer. If it was required to keep only the name of the manufacturer (as in the case with class **Car**) this class should not be necessary. We would have a field of **string** type.

We need a method for adding a car (object of type **Car**) to the list of cars (in **HashSet<Car>**). It will be named **AddSupportedCar(Car car)**.

Below is the code of the class **Part** which is also designed as set of **immutable fields** (except that it accepts adding cars):

Part.cs

```
public class Part
{
   private string name;
   private string code;
   private PartCategory category;
   private HashSet<Car> supportedCars;
   private decimal buyPrice;
```

```
    private decimal sellPrice;
    private Manufacturer manufacturer;

    public Part(string name, decimal buyPrice, decimal sellPrice,
      Manufacturer manufacturer, string code,
      PartCategory category)
    {
      this.name = name;
      this.buyPrice = buyPrice;
      this.sellPrice = sellPrice;
      this.manufacturer = manufacturer;
      this.code = code;
      this.category = category;
      this.supportedCars = new HashSet<Car>();
    }

    public void AddSupportedCar(Car car)
    {
      this.supportedCars.Add(car);
    }

    public override string ToString()
    {
      StringBuilder result = new StringBuilder();
      result.Append("Part: " + this.name + "\n");
      result.Append("-code: " + this.code + "\n");
      result.Append("-category: " + this.category + "\n");
      result.Append("-buyPrice: " + this.buyPrice + "\n");
      result.Append("-sellPrice: " + this.sellPrice + "\n");
      result.Append("-manufacturer: " + this.manufacturer +"\n");
      result.Append("---Supported cars---" + "\n");
      foreach (Car car in this.supportedCars)
      {
        result.Append(car);
        result.Append("\n");
      }
      result.Append("----------------------\n");
      return result.ToString();
    }
}
```

In the class **Part** we use **HashSet<Car>** so it is necessary to **redefine the methods Equals(…) and GetHashCode() for the class Car**:

```
// The Equals(…) and GetHashCode() methods for the class Car

public override bool Equals(object obj)
{
   Car otherCar = obj as Car;
   if (otherCar == null)
   {
      return false;
   }
   bool equals =
      object.Equals(this.brand, otherCar.brand) &&
      object.Equals(this.model, otherCar.model) &&
      object.Equals(this.productionYear,otherCar.productionYear);
   return equals;
}

public override int GetHashCode()
{
   const int prime = 31;
   int result = 1;
   result = prime * result + ((this.brand == null) ? 0 :
      this.brand.GetHashCode());
   result = prime * result + ((this.model == null) ? 0 :
      this.model.GetHashCode());
   result = prime * result + this.productionYear;
   return result;
}
```

Testing the Class Part

We test the class **Part**. It is a bit more complicated than when testing the classes **Car** and **Manufacturer**, because **Part** it is more complex class. We can create a part, assign all its properties and print it:

```
Manufacturer bmw = new Manufacturer("BWM",
   "Germany", "Bavaria", "665544", "876666");
Part partEngineOil = new Part("BMW Engine Oil",
   633.17m, 670.0m, bmw, "Oil431", PartCategory.Engine);
Car bmw316i = new Car("BMW", "316i", 1994);
partEngineOil.AddSupportedCar(bmw316i);
Car mazdaMX5 = new Car("Mazda", "MX5", 1999);
partEngineOil.AddSupportedCar(mazdaMX5);
Console.WriteLine(partEngineOil);
```

Seems like the result is **correct**:

```
Part: BMW Engine Oil
-code: Oil431
-category: Engine
-buyPrice: 633.17
-sellPrice: 670.0
-manufacturer: BWM <Germany,Bavaria,665544,876666>
---Supported cars---
<BMW,316i,1994>
<Mazda,MX5,1999>
----------------------
```

Before we can continue with the next class, we could **test for duplicated cars** in the set of supported cars for certain part. Duplicates are not allowed by design and we should check whether this is enforced:

```
Manufacturer bmw = new Manufacturer("BWM",
  "Germany", "Bavaria", "665544", "876666");
Part partEngineOil = new Part("BMW Engine Oil",
  633.17m, 670.0m, bmw, "Oil431", PartCategory.Engine);
partEngineOil.AddSupportedCar(new Car("BMW", "316i", 1994));
partEngineOil.AddSupportedCar(new Car("BMW", "X5", 2006));
partEngineOil.AddSupportedCar(new Car("BMW", "X5", 2007));
partEngineOil.AddSupportedCar(new Car("BMW", "X5", 2006));
partEngineOil.AddSupportedCar(new Car("BMW", "316i", 1994));
Console.WriteLine(partEngineOil);
```

The result is **correct**. The duplicated cars are taken into account only once:

```
Part: BMW Engine Oil
-code: Oil431
-category: Engine
-buyPrice: 633.17
-sellPrice: 670.0
-manufacturer: BWM <Germany,Bavaria,665544,876666>
---Supported cars---
<BMW,316i,1994>
<BMW,X5,2006>
<BMW,X5,2007>
----------------------
```

Step 5: The Class Shop

We already have all needed classes for **creating the class Shop**. It will have two fields: name and list of parts, which are for sale. The list will be **`List<Part>`**. We will add the method **`AddPart(Part part)`**, with which we

will add new parts. With a redefined **ToString()** we will print the name of the shop and the parts in it.

Here is an example of implementation of our class **Shop** holding the catalog of auto parts (its name is immutable but it can add parts):

Shop.cs

```
public class Shop
{
   private string name;
   private List<Part> parts;

   public Shop(string name)
   {
      this.name = name;
      this.parts = new List<Part>();
   }

   public void AddPart(Part part)
   {
      this.parts.Add(part);
   }

   public override string ToString()
   {
      StringBuilder result = new StringBuilder();
      result.Append("Shop: " + this.name + "\n\n");
      foreach (Part part in this.parts)
      {
         result.Append(part);
         result.Append("\n");
      }
      return result.ToString();
   }
}
```

It might be a **subject of discussion whether we should use List<Part> or Set<Part>** for the parts in the car shop. The **set data structure** has an advantage that it **avoids any duplicates**. Thus if we have for example few tires of certain model, they will be found only once in the set. To use set we need to be sure the parts are uniquely identified by their code or by some other unique identifier. In our case we assume we could have parts with exactly the same code, name, etc. which come at different buy and sell prices (e.g. if the prices change over the time). So we need to allow duplicated parts

and thus **using a set will not be appropriate**. Parts in the shop will be kept in **List<Part>**.

We will test the class **Shop** though the especially written class **TestShop**.

Step 6: The Class TestShop

We created all classes we need. We have to create one more, with which we will have **to demonstrate the usage of the rest of the classes**. It will be named **TestShop**. In the **Main()** method we will create two manufacturers and a few cars. We will add them to two parts. We will add the parts to the **Shop**. At the end we will print everything on the console.

TestShop.cs

```
public class TestShop
{
   static void Main()
   {
      Manufacturer bmw = new Manufacturer("BWM",
         "Germany", "Bavaria", "665544", "876666");
      Manufacturer lada = new Manufacturer("Lada",
         "Russia", "Moscow", "653443", "893321");

      Car bmw316i = new Car("BMW", "316i", 1994);
      Car ladaSamara = new Car("Lada", "Samara", 1987);
      Car mazdaMX5 = new Car("Mazda", "MX5", 1999);
      Car mercedesC500 = new Car("Mercedes", "C500", 2008);
      Car trabant = new Car("Trabant", "super", 1966);
      Car opelAstra = new Car("Opel", "Astra", 1997);

      Part cheapPart = new Part("Tires 165/50/R13", 302.36m,
         345.58m, lada, "T332", PartCategory.Tires);
      cheapPart.AddSupportedCar(ladaSamara);
      cheapPart.AddSupportedCar(trabant);

      Part expensivePart = new Part("Universal Car Engine",
         6733.17m, 6800.0m, bmw, "EU33",  PartCategory.Engine);
      expensivePart.AddSupportedCar(bmw316i);
      expensivePart.AddSupportedCar(mazdaMX5);
      expensivePart.AddSupportedCar(mercedesC500);
      expensivePart.AddSupportedCar(opelAstra);

      Shop newShop = new Shop("Tuning Pro Shop");
      newShop.AddPart(cheapPart);
      newShop.AddPart(expensivePart);
```

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

This is **the result** of the execution of the above code:

```
Shop: Tuning Pro Shop

Part: Tires 165/50/R13
-code: T332
-category: Tires
-buyPrice: 302.36
-sellPrice: 345.58
-manufacturer: Lada <Russia,Moscow,653443,893321>
---Supported cars---
<Lada,Samara,1987>
<Trabant,super,1966>
----------------------

Part: Universal Car Engine
-code: EU33
-category: Engine
-buyPrice: 6733.17
-sellPrice: 6800.0
-manufacturer: BWM <Germany,Bavaria,665544,876666>
---Supported cars---
<BMW,316i,1994>
<Mazda,MX5,1999>
<Mercedes,C500,2008>
<Opel,Astra,1997>
----------------------
```

Testing the Solution

At the end we need to **test our code**. In fact we have done this in the class **TestShop**. This doesn't mean that we have tested entirely our problem. We have to **check the border cases**, for example when some of the lists are empty. Let's make a little change of the code in **Main()** method, to start the program with an **empty list**:

```
static void Main()
{
    Shop emptyShop = new Shop("Empty Shop");
    Console.WriteLine(emptyShop);
```

```
    Manufacturer lada = new Manufacturer("Lada",
      "Russia", "Moscow", "653443", "893321");
    Part tires = new Part("Tires 165/50/R13", 302.36m,
      345.58m, lada, "T332", PartCategory.Tires);

    Manufacturer bmw = new Manufacturer("BWM",
      "Germany", "Bavaria", "665544", "876666");
    Part engineOil = new Part("BMW Engine Oil",
      633.17m, 670.0m, bmw, "Oil431", PartCategory.Engine);
    engineOil.AddSupportedCar(new Car("BMW", "316i", 1994));

    Shop ultraTuningShop = new Shop("Ultra Tuning Shop");
    ultraTuningShop.AddPart(tires);
    ultraTuningShop.AddPart(engineOil);

    Console.WriteLine(ultraTuningShop);
}
```

The result of this test is:

```
Shop: Empty Shop


Shop: Ultra Tuning Shop

Part: Tires 165/50/R13
-code: T332
-category: Tires
-buyPrice: 302.36
-sellPrice: 345.58
-manufacturer: Lada <Russia,Moscow,653443,893321>
---Supported cars---
----------------------

Part: BMW Engine Oil
-code: Oil431
-category: Engine
-buyPrice: 633.17
-sellPrice: 670.0
-manufacturer: BWM <Germany,Bavaria,665544,876666>
---Supported cars---
<BMW,316i,1994>
----------------------
```

From the result it seems the first shop is empty and in the second shop the list of cars for the first part is empty. This is the **correct output**. Therefore our program works correctly with the border case of empty lists.

We can continue testing with **other border cases** (e.g. missing part name, missing price, missing manufacturer, etc.), as well as with some kind of **performance test** (e.g. shop with 300,000 parts for 5,000 cars and 200 manufacturers). We will leave this for the readers.

Exercises

1. You are given an input file **mails.txt**, which contains names of users and their email addresses. Each line of the file looks like this:

```
<first name> <last name> <username>@<host>.<domain>
```

   There is a requirement for email addresses – **<username>** can be a sequence of Latin letters (**a**-**z**, **A**-**Z**) and underscore (**_**), **<host>** is a sequence of lower Latin letters (**a**-**z**), and **<domain>** has a limit of 2 to 4 lower Latin letters (**a**-**z**). Following the guidelines for problem solving write a program, **which finds the valid email addresses** and writes them together with the names of the users (in the same format as in the input) to an output file **valid-mails.txt**.

   Sample input file (**mails.txt**):

```
Steve Smith steven_smith@yahoo.com
Peter Miller pm<5.gmail.com
Svetlana Green svetlana_green@hotmail.com
Mike Johnson mike*j@888.com
Larry Cutts larry.cutts@gmail.com
Angela Hurd angel&7@freemail.hut.fi
```

   Output file (**valid-mails.txt**):

```
Steve Smith steven_smith@yahoo.com
Svetlana Green svetlana_green@hotmail.com
Larry Cutts larry.cutts@gmail.com
```

2. You are given a **labyrinth**, which consists of **N x N squares**, and each of them can be passable (**0**) or not (**x**).

   In one of the squares our hero Jack (*****) is positioned. Two squares are **neighbors**, if they have a common wall. At one step Jack can pass from one passable square to its neighboring passable square. Write a program, which prints **the number of possible exits** from given labyrinth. At the figure below we have **7 possible exits**, reachable from the start position.

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

The **input data** is read from a text file named **`Labyrinth.in`**. At the first line in the file is the number **N** (2 < N < 1000). At the next N lines there are N characters, each either "**0**" or "**x**" or "*****". The output is a single number and should be printed in the file **`Labyrinth.out`**.

3. You are given a **labyrinth**, which consists of **N x N squares**, each of it can be passable or not. Passable cells consist of lower Latin letter between "**a**" and "**z**", and the non-passable – '**#**'. In one of the squares is Jack. It is marked with "*****".

   Two squares are **neighbors**, if they have common wall. At one step Jack can pass from one passable square to its neighboring passable square. When Jack passes through passable squares, he writes down the letters from each square. At each exit he **gets a word**. Write a program, which from a given labyrinth prints the words, which Jack gets from all the possible exits. At the example below Jack can get **10 different words** corresponding to its 10 possible paths he could find to some of the exits: *a*, *az*, *aza*, *madk*, *madkm*, *madam*, *madamk*, *dir*, *did*, *difid*.

a	#	#	k	m	#
z	#	a	d	a	#
a	*	m	#	#	#
#	d	#	#	#	#
r	i	f	i	d	#
#	d	#	d	#	t

   The input data is read from a text file named **`Labyrinth.in`**. At the first line in the file there is the number N (2 < N < 10). At each of the next N lines there are N characters, each of them is either Latin letter between "**a**" and "**z**" or "**#**" (impassable wall) or "*****" (Jack). The output must be printed in the file **`Labyrinth.out`**.

4. A company plans to create a **system for managing of a sound recording company**. The sound recording company has a name, address, owner and performers. Each **performer** has name, nickname and created albums. **Albums** are described with name, genre, year of creation, number of sold copies and list of songs. The **songs** are described with name and duration. Design a **set of classes with**

relationships between each other, which models the data of the record company. Implement a test class, which demonstrates the work of rest of the classes.

5. A company plans on **creating of a system for managing a company for real estates**. The company **has** name, owner, tax ID, employees and has a list of estates for sale. **Employees** are described with name, work position and experience. The company sells several **types of estates**: apartments, houses, undeveloped areas and shops. All of them are characterized with area, price of square meters and location. For some of them there is additional information. For the **apartments** there is data about the number of the floor, whether there is an elevator in the block, and if it is furnished. For the **houses** the data is – square meters for the undeveloped area and for the developed (yard), how many floors it has and whether it is furnished. **Design a set of classes with relationships** between them, which model the data for the company. **Implement a test class**, which demonstrates the work of the rest of the classes.

Solutions and Guidelines

1. The problem is similar to the first problem from our sample exam. Again we can read line by line the input file and with appropriate **regular expression** to check the email addresses. **Test the solution** carefully before you go to the next problem.
2. **Possible exits** from the labyrinth are all the cells, which are positioned at the border of the labyrinth and are reachable from the initial cell. The problem could be solved using **BFS** with **just little modification** of the solution of the **"Escape from Labyrinth"**. **Test your solution** carefully!
3. The problem is similar to the previous one, but **all possible paths** to the exit are required. You can do recursive search with **backtracking** (**DFS**) and keep in a **`StringBuilder`** the letters to the exit, to create the words, which you have to print. With bigger labyrinths the problem has no optimal solution (there is no way to print all the paths, without generating all of them, but they **grow exponentially** to the labyrinth size). **Test** carefully your solution and think of special cases that need special care.
4. You must write the required classes: **`MusicCompany`**, **`Artist`**, **`Album`**, **`Song`**. Think of the links between classes and what data structures to use for them. For the printing redefine the method **`ToString()`** from **`System.Object`**. **Test all methods** and the border cases.
5. The classes you must write are **`EstateCompany`**, **`Employee`**, **`Apartment`**, **`House`**, **`Shop`** and **`UndevelopedArea`**. Export all shared characteristics in separate **abstract base class `Estate`**. Encapsulate all fields with properties. Override the method **`ToString()`**, which to collect the data of the corresponding class and print it to the console. **Test all methods** and special border cases (like missing property values).