Chapter 13. Strings and Text Processing

In This Chapter

In this chapter we will explore **strings**. We are going to explain how they are implemented in C# and in what way we can process text content. Additionally, we will go through different methods for **manipulating a text**: we will learn how to compare strings, how to search for substrings, how to extract substrings upon previously settled parameters and last but not least how to split a string by separator chars. We will demonstrate how to **correctly build strings** with the **`StringBuilder`** class. We will provide a short but very useful information for the most commonly used **regular expressions**. We will discuss some classes for efficient construction of strings. Finally, we will take a look at the methods and classes for achieving more elegant and stricter **formatting** of the text content.

Strings

In practice we often come to the **text processing**: reading text files, searching for keywords and replacing them in a paragraph, validating user input data, etc… In such cases we can save the text content, which we will need in strings, and process them using the C# language.

What Is a String?

A string is a **sequence of characters** stored in a certain address in memory. Remember the type **`char`**? In the variable of type **`char`** we can record only one character. Where it is necessary to process more than one character then strings come to our aid.

In. NET Framework each character has a serial number from the **Unicode table**. The Unicode standard is established in the late 80s and early 90s in order to store different types of text data. Its predecessor **ASCII** is able to record only 128 or 256 characters (respective ASCII standard with 7-bit or 8-bit table). Unfortunately, this often does not meet user needs – as we can fit in 128 characters only digits, uppercase and lowercase Latin letters and some specific individual characters. When you have to work with text in Cyrillic or other specific language (e.g. Chinese or Arabian), 128 or 256 characters are extremely insufficient. Here is why .NET uses 16-bit code table for the characters. With our knowledge of number systems and representation of information in computers, we can calculate that the code table store 2^16 =

65,536 characters. Some characters are encoded in a specific way, so it is possible to use two characters of the Unicode table to create a new character – the resulting signs exceed 100,000.

The System.String Class

The class **System.String** enables us to handle **strings in C#**. For declaring the strings we will continue using the **keyword string**, which is an alias in C# of the **System.String** class from .NET Framework. The work with string facilitates us in manipulating the text content: construction of texts, text search and many other operations.

Example of **declaring a string**:

```
string greeting = "Hello, C#";
```

We have just declared the variable **greeting** of type **string** whose content is the text phrase "**Hello, C#**". The representation of the content in the string looks closely to this:

H	e	l	l	o	,		C	#

The internal representation of the class is quite simple – an array of characters. We can avoid the usage of the class by declaring a variable of type **char[]** and fill in the array's elements character by character. However, there are some disadvantages too:

1. Filling in the array happens character by character, not at once.
2. We should know the length of the text in order to be aware whether it will fit into the already allocated space for the array.
3. The text processing is manual.

The String Class: Universal Solution?

The usage of **System.String** is not the ideal and universal solution – sometimes it is appropriate to use different character structures.

In C# we there are other classes for text processing – we will become familiar with some of them later in this chapter.

The type **string** is more special from other data types. It is a class and as such it complies with the principles of object-oriented programming. Its values are stored in the dynamic memory (**managed heap**), and the variables of type **string** keeps a **reference** to an object in the heap.

Strings are Immutable

The **string** class has an important feature – the character sequences stored in a variable of the class are never changing (**immutable**). After being assigned once, the content of the variable does not change directly – if we try

to change the value, it will be saved to a new location in the dynamic memory and the variable will point to it. Since this is an important feature, it will be illustrated later.

Strings and Char Arrays

Strings are very similar to the char arrays (**char[]**), but unlike them, they **cannot be modified**. Like the arrays, they have properties such as **Length**, which returns the length of the string and allows access by index. Indexing, as it is used in arrays, takes indices from **0** to **Length-1**. Access to the character of a certain position in a string is done with the operator **[]** (indexer), but it is allowed only to read characters (and not to write to them):

```
string str = "abcde";
char ch = str[1]; // ch == 'b'
str[1] = 'a'; // Compilation error!
ch = str[50]; // IndexOutOfRangeException
```

Strings – Simple Example

Let's give an example for using variables from the type **string**:

```
string message = "This is a sample string message.";

Console.WriteLine("message = {0}", message);
Console.WriteLine("message.Length = {0}", message.Length);

for (int i = 0; i < message.Length; i++)
{
  Console.WriteLine("message[{0}] = {1}", i, message[i]);
}
// Console output:
// message = This is a sample string message.
// message.Length = 31
// message[0] = T
// message[1] = h
// message[2] = i
// message[3] = s
// …
```

Please note the string value – the quotes are not part of the text, they are enclosing its value. The example demonstrates how to print a string, how to extract its length and how to extract the character from which it is composed.

Strings Escaping

As we already know, if we want to use quotes into the string content, we must put a slash before them to identify that we consider the quotes character itself and not using the quotation marks for ending the string:

```
string quote = "Book's title is \"Intro to C#\"";
// Book's title is "Intro to C#"
```

The quotes in the example are part of the text. They are added in the variable by placing them after the escaping character backslash (****). In this way the compiler recognizes that the quotes are not used to start or end a string, but are a part of the data. Displaying special characters in the source code is called **escaping**.

Declaring a String

We can declare variables from the type string by the following rule:

```
string str;
```

Declaring a string represents a variable declaration of type **`string`**. This is not equivalent to setting a variable and allocating memory for it! With the declaration we inform the compiler that the variable **`str`** will be used and the expected type for it is **`string`**. We do not create a variable in the memory and it is not available for processing yet (value is **`null`**, which means no value).

Creating and Initializing a String

In order to process the declared string variable, we must create it and initialize it. Creating a variable of certain class (also known as **instantiating**) is a process associated with the allocation of the dynamic memory area (the heap). Before setting a specific value to the string, its value is **`null`**. This can be confusing to the beginner programmers: uninitialized variables of type **`string`** do not contain empty values, it contains the special value **`null`** – and each attempt for manipulating such a string will generate an error (exception for access to a missing value **`NullReferenceException`**)!

We can initialize variables in the following three ways:

1. By assigning a string literal.
2. By assigning the value of another string.
3. By passing the value of an operation which returns a string.

Setting a String Literal

Setting a string literal means **to assign a predefined textual content** to a variable of type **`string`**. We use this type of initialization, when we know the value that must be stored in the variable. Example for setting a string literal:

```
string website = "http://www.wikipedia.org";
```

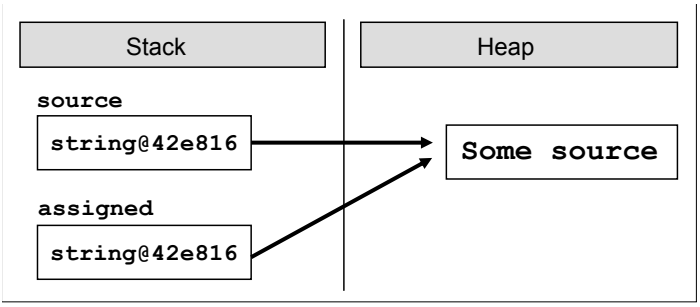
In this example we created the variable **website** with value the above stated string literal.

Assigning Value of Another String

Assigning a value is equivalent to directing a **string** value or a variable to a variable of type **string**. An example is the following code snippet:

```
string source = "Some source";
string assigned = source;
```

First, we declare and initialize the variable **source**. Then the variable **assigned** takes the value of **source**. Since the **string** class is a **reference type**, the text "**Some source**" is stored in the dynamic memory (heap) on an address defined by the first variable.



In the second line we redirect the variable **assigned** to the same place, which the other variable points to. In this way the two objects receive the same address in dynamic memory and hence the same value.

The change of either variable will affect **only itself** because of the immutability of the type **string**, as when a change occurs, a copy of the changed string will be created. This is not true for the rest of the reference types (the normal, mutable types) because with them the changes are made directly in the address in memory and all references point to this changed address.

Passing a String Expression

The third option to initialize a string is to **pass the value of a string expression** or operation, which returns a string result. This can be a result from a method, which validates data; adding together the values of a number of constants and variables; transforming an existing variable, etc.

Example of an expression, which returns a string:

```
string email = "some@gmail.com";
```

```
string info = "My mail is: " + email;
// My mail is: some@gmail.com
```

The **info** variable has been created from the **concatenation** of literals and a variable.

Reading and Printing to the Console

Let's now take a look at the ways of reading strings, entered by the user and how we print strings to the console.

Reading Strings

Reading strings can be accomplished through the methods of the well-known **System.Console** class:

```
string name = Console.ReadLine();
```

In this example we read from the console the input data through the method **ReadLine()**. It waits for the user to input a value and to press [Enter]. After pressing the [Enter] key the variable **name** will contain the input name typed at the console (read from the keyboard).

What can we do after the variable has been created and it has a value itself? We can use it, for example, in expressions with other strings, to pass it as a method's parameter, to write it in text documents, etc. First, we can write it to the console in order to be sure that the data has been correctly read.

Printing Strings

Taking the data to the standard output is made also by the well-known class **System.Console**:

```
Console.WriteLine("Your name is: " + name);
```

By using the method **WriteLine(…)** we are getting the message "**Your name is:** " followed by the value of the **name** variable. After the end of the message a new line character is added. If we want to run away from the new line, which means the messages will appear at one and the same line then we use the method, **Write(…)**.

We can refresh our knowledge on the **System.Console** class from the chapter "Console Input and Output".

Strings Operations

After getting familiar with the strings semantics and how we can create and print them, next comes to learn how to deal with them and how to process

them. The C# language gives us a number of operations ready for use, which we will use for manipulating the strings.

Comparing Strings in Alphabetical Order

There are **many ways to compare strings** and depending on what exactly we need in the particular case, we can take advantage of the various features of the **string** class.

Comparison for Equality

If the requirements are to **compare the two strings** in order to determine whether their values are **equal or not**, the most convenient method is the **Equals(…)**, which works equivalently to the **operator ==**. It returns a Boolean result with either **true** value, if the strings have the same values, or **false** value, if they are different. The method **Equals(…)** checks letter by letter for equality of string values, as it makes distinction between small and capital letters, i.e. comparing the "c#" and "C#" with the **Equals(…)** method will return the value **false**. Consider the following example:

```
string word1 = "C#";
string word2 = "c#";
Console.WriteLine(word1.Equals("C#"));
Console.WriteLine(word1.Equals(word2));
Console.WriteLine(word1 == "C#");
Console.WriteLine(word1 == word2);

// Console output:
// True
// False
// True
// False
```

In practice, we often are interested of only the actual text content when comparing two strings, regardless of the **character casing** (uppercase / lowercase). To ignore the difference between small and capital letters in string comparison we can use the method **Equals(…)** with the parameter **StringComparison.CurrentCultureIgnoreCase**. So now in the same example of comparing "C#" with "c#" the method will return the value **true**:

```
Console.WriteLine(word1.Equals(word2,
  StringComparison.CurrentCultureIgnoreCase));
// True
```

StringComparison.CurrentCultureIgnoreCase is a constant of the enumerated type **StringComparison**. What is enumerated type and how we can use it, we will learn in the chapter "Defining Classes".

Comparing Strings in Alphabetical Order

It has become clear how we compare strings for equality, but how we are going to establish the **lexicographical order** of several strings? If we try to use the operators **<** and **>** which work great for comparing numbers, we find out that they **cannot be used for strings**.

If you want to compare two words and get information which one of them is before the other according to their **alphabetical order** of letters, here comes the method **CompareTo(…)**. It allows us to compare the values of two strings in order to determine their lexicographical order. In order two strings to have the same values, they must have the same length (number of characters) and the all their characters should match accordingly. For example, the strings "give" and "given" are different because they differ in their lengths, and "near" and "fear" differ in their first character.

The method **CompareTo(…)** from the **String** class returns a negative value, 0 or positive value depending on the lexical order of the two compared strings. A negative value means that the first string is **lexicographically** before the second, zero means that the two strings are equal and positive value means that the second string is lexicographically before the first. To clarify better how to compare strings lexicographically, let's go through a few examples:

```
string score = "sCore";
string scary = "scary";

Console.WriteLine(score.CompareTo(scary));
Console.WriteLine(scary.CompareTo(score));
Console.WriteLine(scary.CompareTo(scary));

// Console output:
// 1
// -1
// 0
```

The first experiment is called the method **CompareTo(…)** of the string **score**, as passed parameter is the variable **scary**. The first digit returns equal sign. Because the method **does not ignore** the casing of small and capital letters, it finds mismatch in the second character (in the first string it is "C", while in the second it is "c"). This is enough to determine the arrangement of strings and **CompareTo(…)** returns +1. Calling the same method with swapped places of the strings returns -1, because then the starting point is the string **scary**. His final call returns a logical 0, because we compare **scary** with itself.

If we have to **compare the strings lexicographically**, namely to **ignore the letters casing**, then we could use **string.Compare(string strA, string strB, bool ignoreCase)**. This is a static method, which works in the same way as **CompareTo(…)**, but it has an **ignoreCase** option for ignoring the casing of capital and small letters. Let's look at the method in action:

```
string alpha = "alpha";
string score1 = "sCorE";
string score2 = "score";

Console.WriteLine(string.Compare(alpha, score1, false));
Console.WriteLine(string.Compare(score1, score2, false));
Console.WriteLine(string.Compare(score1, score2, true));
Console.WriteLine(string.Compare(score1, score2,
  StringComparison.CurrentCultureIgnoreCase));
// Console output:
// -1
// 1
// 0
// 0
```

In the last example the method **Compare(…)** takes as a third parameter **StringComparison.CurrentCultureIgnoreCase** – already well-known from the method **Equals(…)** through which we can also compare strings, without having to register the difference between the small and capital letters.

Please note that according to the methods **Compare(…)** and **CompareTo(…)** the **small letters are lexicographically before the capital ones**. The correctness of this rule is quite controversial as in the Unicode table the capital letters are before the small ones. For example due to the standard Unicode, the letter "**A**" has a code 65, which is smaller than the code of the letter "**a**" (97).

When you want just to consider whether the values of two strings are equal or not, please use the method Equals(…) or the operator ==. The methods CompareTo(…) and string.Compare(…) are designed to be used when the lexicographical order is needed.

Therefore, you should consider that the **lexicographical comparison does not follow the letter arrangement in the Unicode table**. Other abnormalities can also be caused by special features of the current culture. For some languages like German the characters "**ss**" and "**ß**" are considered equal. For example the words "Straße" and "Strasse" are considered the same by **CompareTo(…)** and equal when compared through the **==** operator:

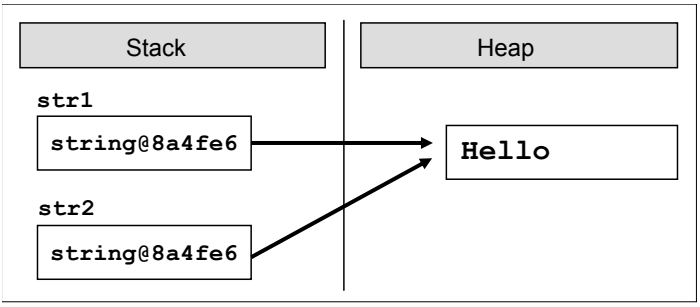
```
string first = "Straße";
string second = "Strasse";

Console.WriteLine(first == second); // False
Console.WriteLine(first.CompareTo(second)); // 0 – equal strings
```

The == and != Operators

In the C# language the operators == and =! work for strings through an internal calling of **Equals(…)**. We will go through some examples for using those two operators with variables from the string type:

```
string str1 = "Hello";
string str2 = str1;

Console.WriteLine(str1 == str2);
// Console output:
// True
```

The comparison of matching strings **str1** and **str2** returns **true**. This is a fully expected result, since the target variable **str2** is pointed to the dynamic memory that is reserved for the variable **str1**. Thus, both variables have the same address and the check for equality returns true. Presented is how the memory looks like with the two variables:

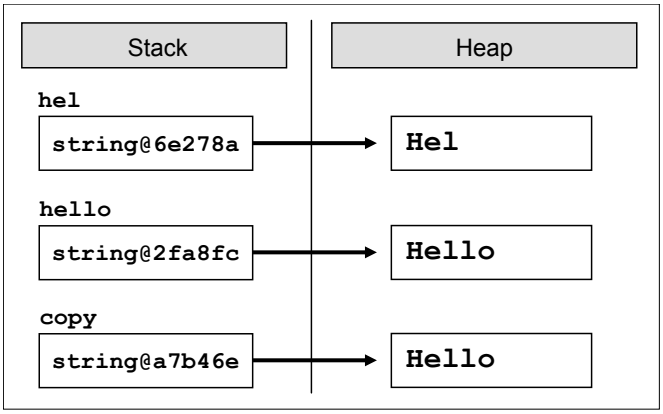


Let's look at another example:

```
string hel = "Hel";
string hello = "Hello";
string copy = hel + "lo";

Console.WriteLine(copy == hello);
// True
```

Pay attention to the **comparison** between the strings **hello** and **copy**. The first variable takes directly the value "**Hello**". The second takes its value as a result of joining a variable with literal, and the final result is equivalent to the value of the first variable. At this stage the two variables point to different areas of memory, but the contents of the memory blocks are identical. The comparison made with the operator == returns a result **true**, although both variables point to different areas of memory.

Here is how the memory looks like at this point:



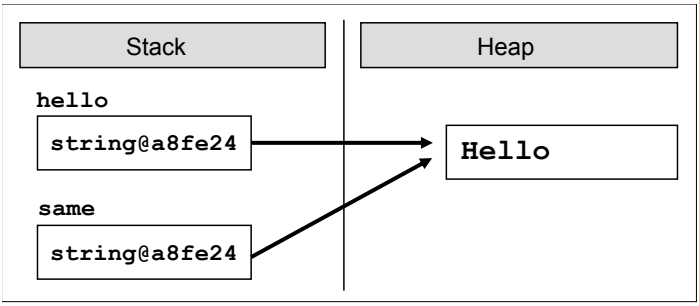
Memory Optimization for Strings (Interning)

Let's consider the following example:

```
string hello = "Hello";
string same = "Hello";
```

Let's create a variable with value "**Hello**". We also create a second variable assigning it a value the same literal. It is logical when creating the variable **hello**, to allocate space in the heap, to write its value and the variable to point to that location. When creating the **same** a new place to record should be allocated too, the value should be written and the reference to the memory should be directed.

But the truth is that there is an optimization in the C# compiler and in CLR, which **saves the memory from creating duplicated strings**. This optimization is called **strings interning** and thanks to it the two variables in the memory will be pointed to the same common block of memory. This reduces the memory space usage and optimizes certain operations such as comparing two completely matching strings. They are written in the memory in the following way:



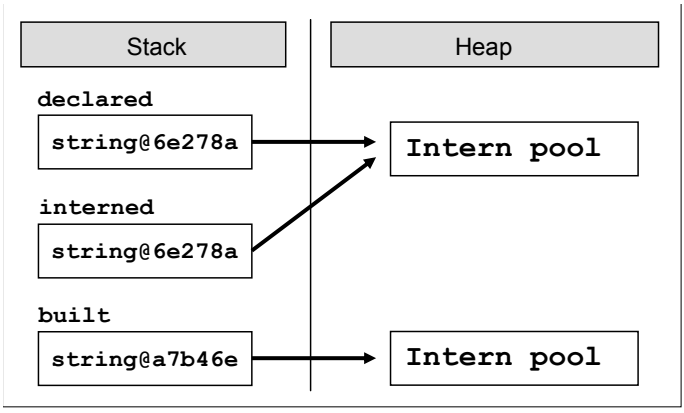
When we initialize a variable of type **string** with a string literal, the memory checks invisibly for us whether this value already exists. If the value already exists, the new variable is simply pointed to it. If not, a new block of memory is allocated, the value is stored in it and the reference is changed to point to

the new block. The **string interning** in .NET is possible because strings are **immutable by design** and it is not likely that the memory block referenced by several string variables will simultaneously be changed by someone.

When not initializing the strings with literals, no interning is used. However, if we want to use interning specifically, we can make it through the use of the method **`Intern(…)`**:

```
string declared = "Intern pool";
string built = new StringBuilder("Intern pool").ToString();
string interned = string.Intern(built);

Console.WriteLine(object.ReferenceEquals(declared, built));
Console.WriteLine(object.ReferenceEquals(declared, interned));
// Console output:
// False
// True
```

Here is the memory situation at this moment:



In the example we used the static method **`Object.ReferenceEquals(…)`**, which compares two objects in memory and returns whether they point to the same memory block. We used the class **`StringBuilder`**, which serves to efficiently build strings. When and how to use **`StringBuilder`** we will explain in details shortly, but now let's get familiar with the basic operations on strings.

Operations for Manipulating Strings

Once we got familiar with the fundamentals of strings and their structure, the next thing to explore are the tools for their processing. We will review string **concatenation**, **searching** in a string, extracting **substrings**, change the character **casing**, **splitting** a string by separator and other string operations that will help us solve various problems from the everyday practice.

Strings are immutable! Any change of a variable of type `string` creates a new string in which the result is stored. Therefore, operations that apply to strings return as a result a reference to the result.

It is possible to process strings without creating new objects in the memory every time a modification is made but for this purpose the class **`StringBuilder`** should be used. We will introduce it a bit later.

Strings Concatenation

Gluing two strings and obtaining a new one as a result is called **concatenation**. It could be done in several ways: through the method **`Concat(…)`** or with the operators **+** and **+=**.

Example of using the method **`Concat(…)`**:

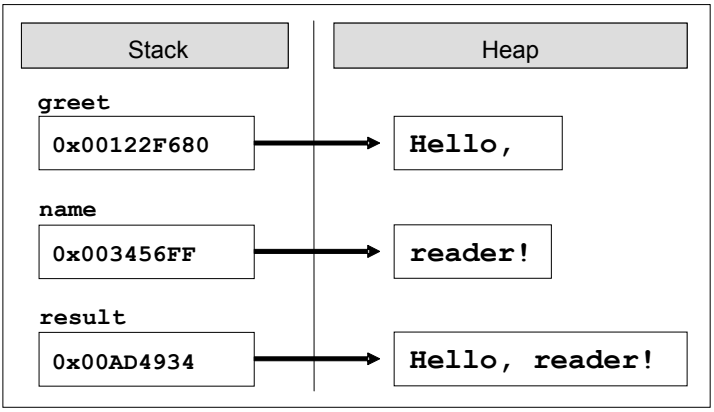
```
string greet = "Hello, ";
string name = "reader!";
string result = string.Concat(greet, name);
```

By calling the method, we will concatenate the string variable **`name`**, which is passed as an argument, to the string variable **`greet`**. The result string will be the text "**Hello, reader!**".

The second way for concatenation is via the operators **+** and **+=**. Then the above example can be implemented in the following way:

```
string greet = "Hello, ";
string name = "reader!";
string result = greet + name;
```

In both cases those variables will be presented in the memory as follows:



Please note that string concatenation **does not change the existing strings** but returns a new string as a result. If we try to concatenate two strings without storing them in a variable, the changes would not be saved. Here is a typical mistake:

```
string greet = "Hello, ";
string name = "reader!";
string.Concat(greet, name);
```

In the given example the two variables are concatenated but the result of it has not been saved anywhere, so it is lost:

If we want to add a value to an existing variable, for example the variable **result**, we can do it with the well-known code:

```
result = result + " How are you?";
```

In order to avoid the double writing of the above declared variable, we can use the operator **+=**:

```
result += " How are you?";
```

The result will be the same in both cases: "**Hello, reader! How are you?**".

We can **concatenate other data with strings**. Any data, which can be presented in a text form, can be appended to a string. Concatenation is possible with numbers, characters, dates, etc. Here is an example:

```
string message = "The number of the beast is: ";
int beastNum = 666;
string result = message + beastNum;
// The number of the beast is: 666
```

As we understood from the above example, there is no problem in concatenating strings with other data, which is not from a **string** type. Let's have another full example for string concatenation:

```
public class DisplayUserInfo
{
  static void Main()
  {
    string firstName = "John";
    string lastName = "Smith";
    string fullName = firstName + " " + lastName;

    int age = 28;
    string nameAndAge = "Name: " + fullName + "\nAge: " + age;
```

```
        Console.WriteLine(nameAndAge);
    }
}
// Console output:
// Name: John Smith
// Age: 28
```

Switching to Uppercase and Lowercase Letters

Sometimes we need to **change the casing of a string** so that all the characters in it to be entirely **uppercase or lowercase**. The two methods that would work best in this case are **`ToLower(…)`** and **`ToUpper(…)`**. The first converts all capital letters to small ones:

```
string text = "All Kind OF LeTTeRs";

Console.WriteLine(text.ToLower());
// all kind of letters
```

The example shows that all capital letters of the text change their casing and the entire text goes in **lowercase**. Such a shift to lowercase is convenient for storing usernames in various online systems. Upon registration the users may use a mixture of uppercase and lowercase letters, but the system can then make them all small to unify them and to avoid matches on points with differences in the casing.

Here is another example. We want to compare entered by the user input but we are not sure exactly how it was written – in small or capital letters or mixed. One possible approach is to standardize capitalization and compare it with the constant defined by us. Thus, we **make no distinction of small and capital letters**. For example, if we have a user input panel where we enter name and password and it does not matter if the password is written with capital letters or small, we can make a similar check on the password:

```
string pass1 = "PasswoRd";
string pass2 = "PaSSwoRD";
string pass3 = "password";

Console.WriteLine(pass1.ToUpper() == "PASSWORD");
Console.WriteLine(pass2.ToUpper() == "PASSWORD");
Console.WriteLine(pass3.ToUpper() == "PASSWORD");

// Console output:
// True
```

```
// True
// True
```

In the example we are comparing three passwords with the same content but with a different casing. When checking their contents, always verify if it equals to the string "**PASSWORD**" (letter by letter). Of course, we could do the above verification and by the method **Equals(…)** in the version with ignoring the character casing, which we already discussed.

Searching for a String within Another String

When we have a string with a specified content, it is often necessary to process only a part of its value. The .NET platform provides us with two methods to **search a string within another string**: **IndexOf(…)** and **LastIndexOf(…)**. They search into the string and check whether the passed as a parameter substring occurs in its content. The result of those methods is an integer. If the result is not a negative value, then this is the position where the first character of the substring is found. If the method returns value of -1, it means that the substring was not found. Remember that in C# indexing into strings start from 0.

The methods **IndexOf(…)** and **LastIndexOf(…)** search the contents of the text sequence, but in a different direction. The search with the first method starts from the beginning of the string towards the end, while the second method – the search is done backwards. If we are interested in the first encountered match, then we use **IndexOf(…)**. If we want to search the string from its end (for example to detect the last dot in a file name or the last slash in an URL address), then we use **LastIndexOf(…)**.

When calling **IndexOf(…)** and **LastIndexOf(…)** a second parameter could be passed, which will specify the position, which the searching should start from. This is useful if we want to search part of a string, not the entire string.

Searching into a String – Example

Let's consider an example with the **IndexOf(…)** method:

```
string book = "Introduction to C# book";
int index = book.IndexOf("C#");

Console.WriteLine(index);
// index = 16
```

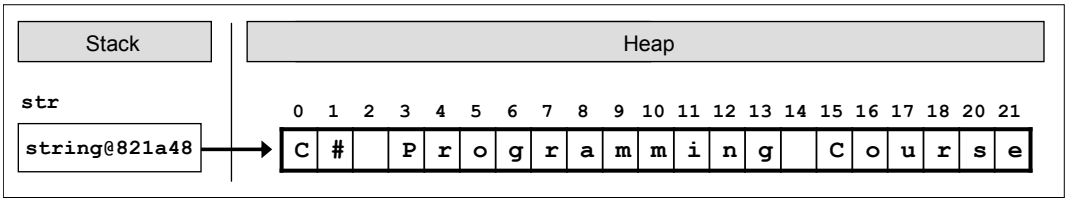
In the example, the variable **book** has a value "**Introduction to C# book**". The search for the substring "**C**" in this variable will return the value 16, because the substring will be found and the first character "**C**" of the searched word is in 16th position.

Searching with IndexOf(…) – Example

Let's look into great details one more example for searching for a separate characters or strings in a text:

```
string str = "C# Programming Course";

int index = str.IndexOf("C#"); // index = 0
index = str.IndexOf("Course"); // index = 15
index = str.IndexOf("COURSE"); // index = -1
index = str.IndexOf("ram");    // index = 7
index = str.IndexOf("r");      // index = 4
index = str.IndexOf("r", 5);   // index = 7
index = str.IndexOf("r", 10);  // index = 18
```

Look how the string we are searching looks like in the memory:



If we look at the results of the third search, we will note that the search for the word "**COURSE**" in the text returned a result of -1, i.e. no match has been found. Although the word is in the text, it has been written in a different case of letters. The methods **IndexOf(…)** and **LastIndexOf(…)** distinguish between uppercase and lowercase letters. If we want to ignore this difference, we can write text in a new variable and turn it to a text with entirely lower or entirely uppercase, and then we can perform the search in it, independently from the letters casing.

Finding All Occurrences of a Substring – Example

Sometimes we want to **find all occurrences of a particular substring within another string**. Using both methods with only one searched string passed as an argument would not work for us, because it will always return only the first occurrence of the substring. We can pass a second parameter for an index that indicates the starting position from which the searching should begin. Of course, we need to loop through it in order to move from the first occurrence of the searched string to the next, to the next, and the next, etc., until the last one.

Here is an example how we can use the method **IndexOf(…)** by a given word and start index: finding all occurrences of the word "C#" in a given text:

```
string quote = "The main intent of the \"Intro C#\"" +
  " book is to introduce the C# programming to newbies.";
```

```
string keyword = "C#";
int index = quote.IndexOf(keyword);

while (index != -1)
{
   Console.WriteLine("{0} found at index: {1}", keyword, index);
   index = quote.IndexOf(keyword, index + 1);
}
```

The first step is to make a search for the keyword "C#". If the word is found in the text (i.e. the returned value is different than -1), it prints it on the console and we continue our search rightwards, starting from the position on which we have found the word plus one. We repeat this operation until **`IndexOf(…)`** returns value -1.

Note: If we miss setting an initial index, then the search will always start from the beginning and will return one and the same value. This will lead to **hanging of the program**. If we search directly from the index without adding plus one each time, we will come across again and again to the last result, whose index we have already found. Therefore, proper search of the next result should start from a starting position **`index + 1`**.

Extracting a Portion of a String

For now we know how to check whether a substring occurs in a text and which are the occurrence positions. But how can we **extract a portion of a string** in a separate variable?

The solution of this problem is the method **`Substring(…)`**. By using it, we can extract a **part of the string (substring)** by a given starting position in the text and its length. If the length is omitted, a portion from the text will be extracted, starting from the initial position to the string's end.

Presented is an example of **extracting a substring from a string**:

```
string path = "C:\\Pics\\CoolPic.jpg";
string fileName = path.Substring(8, 7);
// fileName = "CoolPic"
```

We manipulate the variable **`path`**. It contains the path to a file from our file system. To assign the file name to a new variable, we use **`Substring(8, 7)`** and take a sequence of 7 characters starting from the 8th position, i.e. character positions from 8 to 14 inclusively.

Calling the method `Substring(startIndex, length)`, extracts a substring from a string, which is located between `startIndex` and (`startIndex + length – 1`) inclusively. The character at

	the position `startIndex + length` is not taken into consideration! For example, if we point `Substring(8, 3)`, the characters between index 8 and 10 inclusively will be extracted.

Here are presented the characters, which form the text from which we extract a substring:

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
C	**:**	****	**P**	**i**	**c**	**s**	****	**C**	**o**	**o**	**l**	**P**	**i**	**c**	**.**	**j**	**p**	**g**

Sticking to the scheme, the method that has been called must write the characters from the positions 8 to 14 (as the last index is not included), namely "**CoolPic**".

Extracting a File Name and File Extension – Example

Let's consider a more interesting task. How can we print the **filename and its extension** from given full path to a file in Windows-based file system? As we know how the path is recorded in the file system, we can proceed with the following plan:

- Looking for the **last backslash** in the text;
- Keeping the **position** of the last backslash;
- Extracting the substring starting from **the obtained position +1**;

Let's consider again the example of the well-known **file path**. If we have no information about the exact contents of the variable, but we know that it contains a file path, we can stick to the above scheme:

```
string path = "C:\\Pics\\CoolPic.jpg";
int index = path.LastIndexOf("\\");
// index = 7
string fullName = path.Substring(index + 1);
// fullName = "CoolPic.jpg"
```

Splitting the String by a Separator

One of the most flexible methods for working with strings is **`Split(…)`**. It allows us to **split a string by a separator** or an array of possible separators. For example, we can process a variable, which has the following content:

```
string listOfBeers = "Amstel, Heineken, Tuborg, Becks";
```

How can we split each beer in a separate variable or extract all beers in an array? At first glance it may seem difficult – we must seek with **`IndexOf(…)`** for a special character, then to extract a substring with **`Substring(…)`**, to

iterate all this in a loop and to write the result in a variable. Since the splitting of a string by a separator is a main task of text processing, ready to use methods for it can be found in .NET Framework.

Splitting Strings by Multiple Separators – Example

The easiest and more flexible method for resolving this issue is the following:

```
char[] separators = new char[] {' ', ',', '.'};
string[] beersArr = listOfBeers.Split(separators);
```

Using the built-in functionality of the method **Split(…)** from the class **String**, we will split the contents of a given string by array of characters – separators, which are passed as an argument of the method. All substrings among which are space, comma or dot will be removed and stored in the **beersArr** array.

If we iterate the array and print its elements one by one, the result will be: "Amstel", "", "Heineken", "", "Tuborg", "" and "Becks". We get 7 results, instead of the expected 4. The reason is that during the text splitting, three substrings are found which contain two separator characters one next to the other (for example a comma, followed by a space). In this case the empty string between the two separators is also part of the returned result.

How to Remove the Empty Elements after Splitting?

If we want to ignore the empty strings from the splitting results, one possible solution is to make checks on their printing:

```
foreach (string beer in beersArr)
{
  if (beer != "")
  {
    Console.WriteLine(beer);
  }
}
```

But this approach does not remove the empty strings from the array. It just does not print them. So we can change the arguments we are passing to the method **Split(…)**, by passing a special option:

```
string[] beersArr = listOfBeers.Split(
  separators, StringSplitOptions.RemoveEmptyEntries);
```

After this change, the **beersArr** array will contain 4 elements – the 4 words from the **listOfBeers** variable.

When splitting strings and adding as a second parameter the constant `StringSplitOptions.RemoveEmptyEntries` we instruct

	the method `Split(…)` to work in the following way: "Return all substrings from the variable that are split by given list of separators. If you meet two or more neighboring separators, consider them as one."

Replacing a Substring

The text processing in .NET Framework provides ready methods for **replacing a substring with another**. For example, if we have made one and the same technical mistake when typing the email address of a user in an official document, we can replace it by using the method **`Replace(…)`**:

```
string doc = "Hello, some@gmail.com, " +
  "you have been using some@gmail.com in your registration.";
string fixedDoc =
  doc.Replace("some@gmail.com", "john@smith.com");
Console.WriteLine(fixedDoc);

// Console output:
// Hello, john@smith.com, you have been using
// john@smith.com in your registration.
```

As it can be seen from the example, the method **`Replace(…)` replaces all occurrences** of a given substring with another substring, not just the first.

Regular Expressions

The regular expressions are a powerful tool for text processing and allow searching matches by a **pattern**. An example for a pattern is **`[A-Z0-9]+`**, which means not an empty series of capital Latin letters and numbers.

Regular expressions make text processing **easier and more accurate**: extracting some resources from texts, searching for phone numbers, finding email addresses in a text, splitting all the words in a sentence, data validation, etc.

Regular Expressions – Example

If we have an official document that is used only in the office and it contains a lot of personal data, then we should censor it before sending it to the client. For example, we can censor all mobile numbers and replace them with asterisks. By using **regular expressions**, this could be done as follows:

```
string doc = "Smith's number: 0898880022\nFranky can be " +
  "found at 0888445566.\nSteven's mobile number: 0887654321";
string replacedDoc = Regex.Replace(
  doc, "(08)[0-9]{8}", "$1********");
```

```
Console.WriteLine(replacedDoc);

// Console output:
// Smith's number: 08********
// Franky can be found at 08********.
// Steven' mobile number: 08********
```

Explaining the Arguments of Regex.Replace(…)

In the above code fragment by using a regular expression, we find all the phone numbers specified in the text and replace them by a pattern. We use the class **System.Text.RegularExpressions.Regex**, which is intended for use with regular expressions in .NET Framework. The variable, which imitates the document text, is **doc**. Several names of customers are recorded there. If we want to protect the contacts from an improper use and wish to censor the phone numbers, then we can replace all mobile phones with asterisks. Assuming that the phones are saved in the following format: "**08 + 8 digits**", the method **Regex.Replace(…)** finds all matches by a given format and replaces them with: "**08********"**.

The regular expression that finds all of the numbers is the following: "**(08)[0-9]{8}**". It finds all substrings in the text, constructed by the constant "**08**" and followed exactly by 8 characters ranging from 0 to 9. The example can be further improved by selecting the numbers only from a given mobile operator, for phones on foreign networks, etc., but in this case we used the simplified version.

The literal "**08**" is surrounded by parentheses. They serve for forming a separate group in the regular expression. The groups can be used for handling only a certain part of the expression instead of the entire expression. In our example, the group is used in the substitution. Through it, the founded matches are replaced by the pattern "**$1********"**, i.e. the text which was found in the first group of the regular expression (**$1**) + 8 consecutive asterisks for censorship. As the defined group is always a constant (08), so the text replaced will always be: **08 ********.**

This chapter is not intended to explain in details **how to use regular expressions in .NET Framework**, as it is a huge and complex field, but only to turn the reader's attention that the regular expressions exist and they are a powerful tool for text processing. Anyone who wants to learn more, can search for articles, books and tutorials in order to learn how to construct regular expressions, how to look for matches, how validation is made, how to make substitutions by patterns, etc. In particular, we recommend you to visit the websites http://www.regular-expressions.info and http://regexlib.com. More information about the classes in .NET Framework for working with regular expressions can be found at: http://msdn.microsoft.com/en-us/library/system.text.regularexpressions.regex%28VS.100%29.aspx.

Removing Unnecessary Characters at the Beginning and at the End of a String

When entering text in a file or to the console, you can find sometimes some "parasitic" spaces (**white-space**) at the beginning or at the end of the text – some other space or a tab that cannot be observed at first glance. This may not be essential but if we do not validate the user data, there would be a problem in terms of checking the contents of the input information. In order to solve this problem we can use the method **`Trim()`**. It is responsible for eliminating (**trimming**) the white spaces at the beginning or at the end of a string. The white spaces can be spaces, tabs, line breaks etc.

Let's assume in the variable **`fileData`** we have read the contents of a file where is written a name of a student. There may have emerged parasitic spaces when writing the text or reversing it from one format to another. In that case the variable will look the following way:

```
string fileData = "   \n\n     David Allen      ";
```

If we print the contents to the console, we get two blank lines followed by some spaces, the requested name and some additional spaces at the end. We can reduce the information just to the required name, in the following way:

```
string reduced = fileData.Trim();
```

When we print the information to the console for the second time, the content will be "**David Allen**", without any unwanted white spaces.

Removing Unnecessary Characters by a Given List

The method **`Trim(…)`** can accept an array of characters, which we want to remove from the string. We can make it in the following way:

```
string fileData = "   111 $  %    David Allen  ### s    ";
char[] trimChars = new char[] {' ', '1', '$', '%', '#', 's'};
string reduced = fileData.Trim(trimChars);
// reduced = "David Allen"
```

Again, we get the desired result "**David Allen**".

Please note that we must list all the characters we want to eliminate, including the empty spaces (spaces, tabs, new line, etc.). Without a `' '` in the array `trimChars`, we would not get the desired result!

If we want to remove the white spaces only at the beginning or in end of the string, we can use the methods **`TrimStart(…)`** and **`TrimEnd(…)`**:

```
string reduced = fileData.TrimEnd(trimChars);
// reduced = "   111 $  %    David Allen"
```

Constructing Strings: the StringBuilder Class

As explained above, strings in C# are immutable. This means that any adjustments applied to an existing string do not change it but return a new string. For example, using methods like **`Replace(…)`**, **`ToUpper(…)`**, **`Trim(…)`** do not change the string, which they are called for. They allocate **a new area in the memory** where the new content is saved. This behavior has many advantages but in some cases can cause performance problems.

Strings Concatenation in a Loop: Never Do This!

Serious **performance problems** may be encountered when trying to concatenate strings in a loop. The problem is directly related to the strings handling and dynamic memory, which is used to store them. To understand why we have **poor performance when concatenating strings in a loop**, we must first consider what happens when using operator "**+**" for strings.

How Does the String Concatenation Works?

We already got familiar with the ways to do string concatenation in C#. Let's now examine **what happens in memory when concatenating strings**. Consider two variables **`str1`** and **`str2`** of type **`string`**, which have values of "Super" and "Star". There are two areas in the heap (dynamic memory) in which the values are stored. The task of **`str1`** and **`str2`** is to keep a reference to the memory addresses where our data is stored. Let's create a variable **`result`** and give it a value of the other two strings by concatenation. A code fragment for creating and defining the three variables would look like this:

```
string str1 = "Super";
string str2 = "Star";
string result = str1 + str2;
```

What will happen with the memory? Creating the variable **`result`** will allocate a new area in dynamic memory, which will record the outcome of the **`str1 + str2`**, which is "**SuperStar**". Then the variable itself will keep the address of the allocated area. As a result we will have three areas in memory and three references to them. This is convenient, but allocating a new area, recording a value, creating a new variable and referencing it in the memory is time-consuming process that would be a problem when repeated many times, typically inside a loop.

Unlike other programming languages, in C# is not necessary to manually dispose the objects stored in memory. There is a special mechanism called a **garbage collector (memory cleaning system)**, which takes care of clearing the unused memory and resources. The garbage collector is

responsible for disposing of objects in dynamic memory when they are no longer used. Creation of many objects containing multiple references in dynamic memory is bad, because it fills memory and then the garbage collector is automatically enforced to start execution. It takes quite some time and **slows the overall performance of the process**. Furthermore, transferring characters from one place to another in memory (when string concatenation is executed) is slow, especially if the strings are long.

Why Concatenating Strings in a Loop is a Bad Practice?

Assume that we have a task to store the numbers from 1 to 20,000 consecutively to each other in a variable of type **`string`**. How can we solve the problem with our already existing knowledge? One of the easiest ways for implementation is to create a variable that stores the numbers and execute a loop from 1 to 20,000 in which each number is concatenated to the variable. Implemented in C#, the solution would look like this:

```
string collector = "Numbers: ";
for (int index = 1; index <= 20000; index++)
{
  collector += index;
}
```

Execution of the above code will loop 20,000 times and after each iteration will add the current index to the **`collector`** variable. **`collector`**'s value after implementation would be: "Numbers: 12345678910111213141516…" (the numbers from 17 to 20,000 are replaced with dots because we don't have the space to write something that long here).

Probably you have not noticed the **delay** in the fragment's execution. Indeed, using concatenation in the loop has **delayed significantly** the normal calculation process. On an average PC (as of January 2012) the loop iteration takes **1-2 seconds**. The user of our program would be very skeptical if he has to wait a few seconds for something so simple such as concatenating the numbers from 1 to 20,000. Moreover, in this case 20,000 is just an example endpoint. What will be the delay if instead of 20,000 the user needs to concatenate numbers to 200,000? Try it!

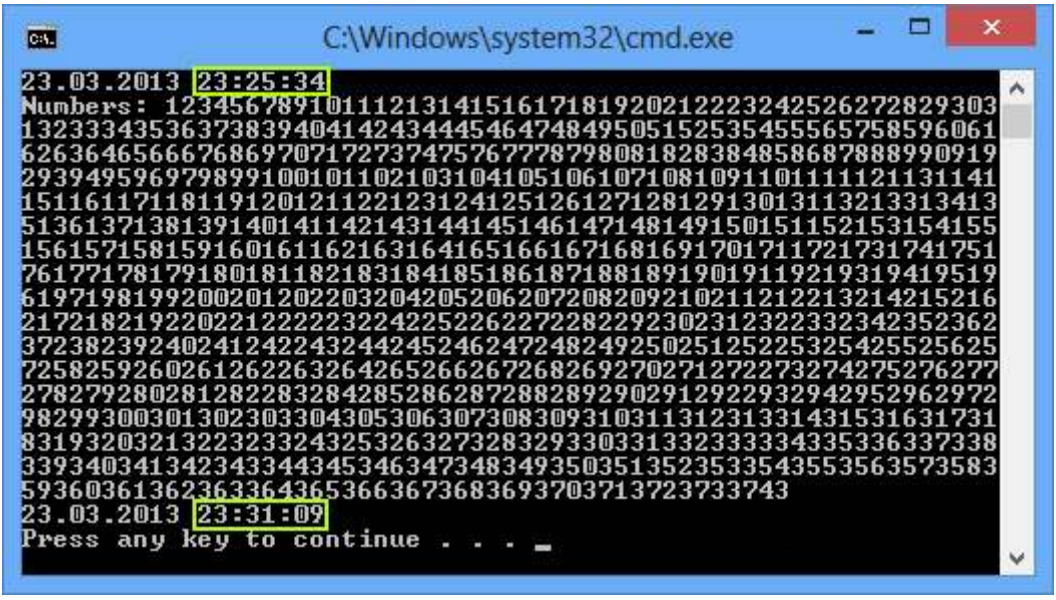
Concatenating in Loop of 200,000 Iterations – Example

Let's develop further the example above. First, we will change the endpoint of the loop from 20,000 to **200,000**. Second, in order to account properly the execution time, we will display on the console the current date and time before and after execution of the loop. Third, to see whether the variable contains the desired value, we will also display part of it on the console. If you want to make sure that the whole value is stored, you can remove the method **`Substring(…)`**, but the print itself in this case will take a long time.

The final version of the example would look like this:

```
class SlowNumbersConcatenator
{
   static void Main()
   {
      Console.WriteLine(DateTime.Now);

      string collector = "Numbers: ";
      for (int index = 1; index <= 200000; index++)
      {
         collector += index;
      }

      Console.WriteLine(collector.Substring(0, 1024));
      Console.WriteLine(DateTime.Now);
   }
}
```

When executing the example implementation on the console, the program starting date and time, the first 1024 characters of the variable and program completion date and time are displayed on the console. The reason to show only the first 1024 characters is that we want to measure only the calculation time without the time for printing the results. Printing the whole result will be time consuming. Let's see sample output from the execution:



Program start is marked with a green line and its end – with red. Note the execution time – about **5-6 minutes** (on our computer from January 2012)! Such a delay is unacceptable for such a task and will not only make the user nervous but will make him stop the program without waiting for it to end.

Processing Strings in the Memory

The problem with time-consuming Loop processing is related to the way strings work in memory. Each iteration creates a new object in the heap and point the reference to it. This process requires a certain physical time.

Several things happen at each step:

1. An area of memory is allocated for recording the next number concatenation result. This memory is used only temporarily while concatenating, and is called a **buffer**.
2. The old string is **moved** into the new buffer. If the string is long (say 500 KB, 5 MB or 50 MB), it can be **quite slow**!
3. Next number is **concatenated** to the buffer.
4. The buffer is **converted to a string**.
5. The old string and the temporary buffer become unused. Later they are **destroyed by the garbage collector**. This may also be a slow operation.

Much more elegant and appropriate way to concatenate strings in a Loop is using the **`StringBuilder`** class. Let's see how it works.

Building and Changing Strings with StringBuilder

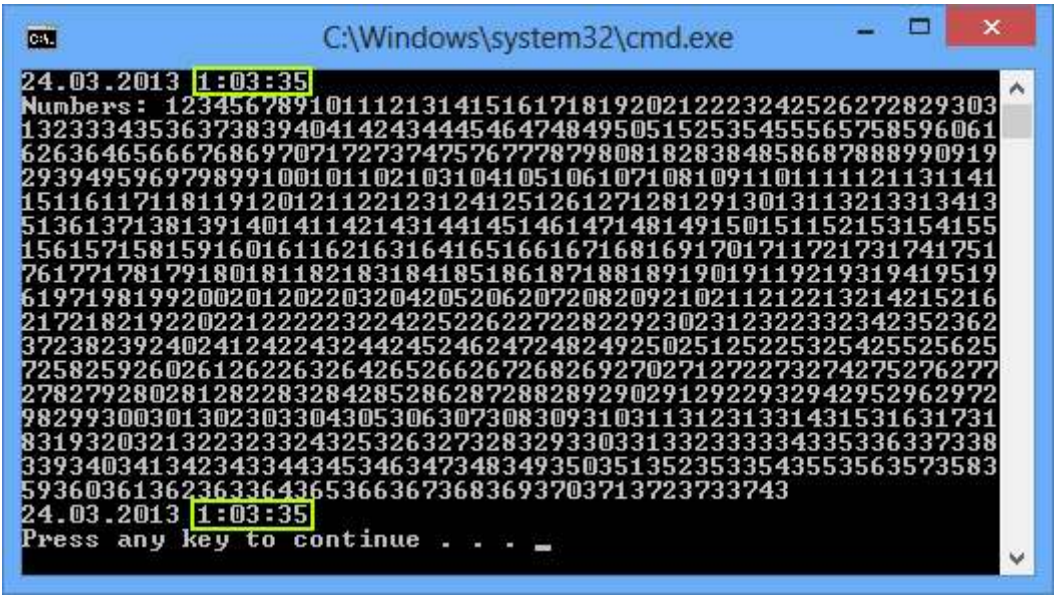
`StringBuilder` is a class that serves to build and change strings. It **overcomes the performance problems** that arise when concatenating strings of type **`string`**. The class is built in the form of an array of characters and what we need to know about it is that the information in it can be freely changed. Changes that are required in the variables of type **`StringBuilder`**, are carried out in the same area of memory (buffer), which saves time and resources. Changing the content does not create a new object but simply changes the current.

Let's rewrite the code above in which we concatenated strings in a loop. If you remember, the operation previously took 5 minutes. Let's measure how long will take the same operation if we use **`StringBuilder`**:

```
class ElegantNumbersConcatenator
{
  static void Main()
  {
    Console.WriteLine(DateTime.Now);

    StringBuilder sb = new StringBuilder();
    sb.Append("Numbers: ");

    for (int index = 1; index <= 200000; index++)
```

```
        {
            sb.Append(index);
        }

        Console.WriteLine(sb.ToString().Substring(0, 1024));
        Console.WriteLine(DateTime.Now);
    }
}
```

This example is based on the previous one, with only minor adjustments. Return value is the same, but what about the execution time?



The time required to concatenate 200,000 characters with **StringBuilder** is now less than a second (perhaps few milliseconds)!

Reversing a String – Example

Consider another example: we want to reverse an existing string (backwards). For example, if we have the string "**abcd**", the returned result should be "**dcba**". We get the original string, iterate it backwards character by character and add each character to a variable of type **StringBuilder**:

```
public class WordReverser
{
    static void Main()
    {
        string text = "EM edit";
        string reversed = ReverseText(text);
```

```
      Console.WriteLine(reversed);

      // Console output:
      // tide ME
   }

   static string ReverseText(string text)
   {
      StringBuilder sb = new StringBuilder();
      for (int i = text.Length - 1; i >= 0; i--)
      {
         sb.Append(text[i]);
      }
      return sb.ToString();
   }
}
```

In this example we have a variable **text**, which contains the value "EM edit". We pass the variable to the **ReverseText(…)** method and set the new value in a variable named **reversed**. The method, in turn, iterates the characters of the variable in reverse order and stores them in a new variable of type **StringBuilder,** but now back ordered. Ultimately, the result is "tide ME".

How Does the StringBuilder Class Work?

The **StringBuilder** class is an implementation of a string in C#, but different than the class **String**. Unlike the already familiar for us strings, the objects of the **StringBuilder** class are not immutable, namely **edit operations do not require creating a new object** in the memory. This reduces the unnecessary transfer of data in memory when performing basic operations such as string concatenation.

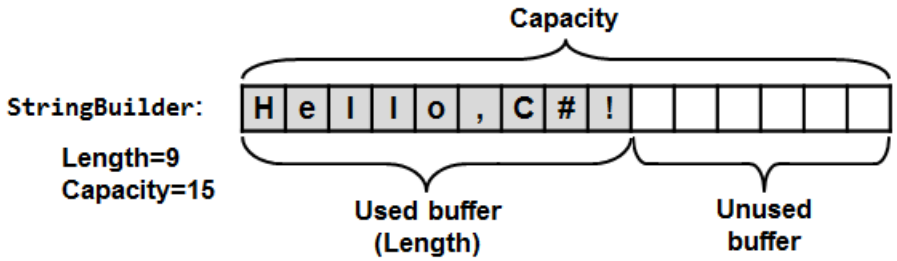
StringBuilder keeps a buffer with a certain capacity (16 characters by default). The buffer is implemented as an array of characters that is provided to the developer by a user-friendly interface – methods that quickly and easily add and edit elements of the string. At any moment part of the characters in the buffer are used and the rest stay in reserve. This allows the addition to work very quickly. Other operations also operate faster than the class **string**, because the changes do not create a new object.

Once the internal buffer of the **StringBuilder** is full, it automatically is doubled (the internal buffer is resized to increase its capacity while its content is kept unchanged). **Resizing is a slow operation** but is happens rarely so the total performance is good. We will discuss this in more details in the chapter about "Algorithms Complexity".

Let's create an object of the **StringBuilder** class with 15 characters long buffer. We add the string: "**Hello, C#!**" to it and we get the following code:

```
StringBuilder sb = new StringBuilder(15);
sb.Append("Hello, C#!");
```

After creating the object and storing the value in it, the **StringBuilder** will look as follows:



Colored elements are the filled with our content part of the buffer. Normally, adding a new character to the variable does not create a new object in the memory but use the already allocated and unused space. If the entire capacity of the buffer is filled, then the buffer is doubled as we already explained.

StringBuilder – More Important Methods

The **StringBuilder** class provides us with a set of methods that help us to easily and efficiently edit text data and construct text. We met some of them in the examples. The most important are:

- **StringBuilder(int capacity)** – constructor with an initial capacity parameter. It may be used to set the buffer size in advance if we have estimates of the number of iterations and concatenations, which will be performed. This way we can save unnecessary dynamic memory allocations.
- **Capacity** – returns the buffer size (total number of used and unused positions in the buffer).
- **Length** – returns length of string saved in the variable (number of used positions in the buffer)
- Indexer **[int index]** – return the character stored in given position.
- **Append(…)** – appends string, number or other value after the last character in the buffer.
- **Clear(…)** – removes all characters from the buffer (deletes it).
- **Remove(int startIndex, int length)** – removes (deletes) string from the buffer with a given start position and length.

- **Insert(int offset, string str)** – inserts a string in a given start position (offset).
- **Replace(string oldValue, string newValue)** – replaces all occurrences of a given substring with another substring.
- **ToString()** – returns the **StringBuilder** object content as a **string** object.

Extracting All Capital Letters from a Text – Example

The next task is to **extract all capital letters from a text**. We can implement it in different ways – using an array, counter and filling the array with all capital letters found; creating an object of type **string** and concatenate capitals one by one to it; using the class **StringBuilder**.

Turning to the option of using an array, we have a problem: we do not know what will be array size, as we have no idea in advance how many are the capital letters in the text. We can create an array as large as the text, but thus wasting unnecessary space in memory and we must also maintain a counter that keeps where the array is full to.

Another option is to use a variable of type **string**. As we will iterate the whole text and concatenate all capital letters to the variable, probably we will lose efficiency again due to the strings concatenation.

StringBuilder: the Right Solution

The most viable solution to the task again is to use **StringBuilder**. We can start with an empty **StringBuilder**, iterate the letters of the given text character by character, verify that the current character is uppercase and concatenate the character at the end of our **StringBuilder**. Finally, we can return the final result by calling the **ToString()** method. Below is a sample implementation:

```
public static string ExtractCapitals(string str)
{
   StringBuilder result = new StringBuilder();
   for (int i = 0; i < str.Length; i++)
   {
      char ch = str[i];
      if (char.IsUpper(ch))
      {
         result.Append(ch);
      }
   }
   return result.ToString();
}
```

Calling **ExtractCapitals(…)** method and passing a specified text as a parameter to it, the return value is a string of all capital letters in the text, namely the initial string without all characters that are not capitalized. To check whether a character is uppercase we are using **char.IsUpper(…)** – a method from the standard .NET classes. You can view the **char** class documentation, because it offers other useful methods for handling characters.

String Formatting

.NET Framework provides the developer with mechanisms for formatting strings, numbers and dates. We have already met some of them in the chapter "Console Input and Output". Now we will extend our knowledge with methods for formatting and converting strings of the **string** class.

The ToString(...) Method

One of the interesting concepts in .NET is that practically every object of a class and primitive variables can be **presented as text**. This is done by the method **ToString(…)**, which is present in all .NET objects. It is implicit in the definition of the **object** class – the base class that all .NET data types inherit directly or indirectly. Thus the definition of the method appears in each class and we can use it to bring the content of each object in some text form.

The method **ToString(…)** is called automatically when we print objects from different classes to the console. For example, when printing dates the submitted date is converted to text by calling the **ToString(…)**:

```
DateTime currentDate = DateTime.Now;
Console.WriteLine(currentDate);
// Output: 01.02.2012 13:34:27 (depends on the culture settings)
```

When we pass **currentDate** as a parameter of the **WriteLine(…)** method, we don't have an accurate statement that handles dates. The method has a particular implementation for all primitive types and strings. For all other objects **WriteLine(…)** calls their **ToString(…)** method, which first converts them to text and then displays the resulting text content. In fact, the sample code above is equivalent to the following:

```
DateTime currentDate = DateTime.Now;
Console.WriteLine(currentDate.ToString());
```

The default implementation of the **ToString(…)** method in the **object** class returns the full name of the class. All classes that do not explicitly redefine the behavior of the **ToString(…)** are using this implementation. Most classes in C# have their own implementation of the method, which represents readable and understandable content in text form. For example, converting a number to text is using the standard format for numbers in the current

culture. Converting a date to text is also using the standard format for dates in the current culture.

Using of String.Format(…)

String.Format(…) is a static method by which we can **format text and other data through a template** (formatting string). The templates contain text and declared parameters (**placeholders**) and are used to obtain formatted text after replacing the parameters with specific values. You can make a direct association with the **Console.WriteLine(…)** method, which also formats a string through a template:

```
Console.WriteLine("This is a template from {0}", "David");
```

How to use the **String.Format(…)** method? Consider an example in order to clarify this:

```
DateTime date = DateTime.Now;
string name = "David Scott";
string task = "Introduction to C# book";
string location = "his office";

string formattedText = String.Format(
  "Today is {0:MM/dd/yyyy} and {1} is working on {2} in {3}.",
  date, name, task, location);
Console.WriteLine(formattedText);

// Output: Today is 01.02.2012 and David Scott is working on
// Introduction to C# book in his office.
```

As it is seen from the example, formatting with **String.Format()** uses **placeholders** (parameters like **{0}**, **{1}**, etc.) and accepts formatting strings (such as **:dd.MM.yyyy**). It accepts as first parameter a formatting string containing text with parameters, followed by values for each parameter and returns the formatted text as a result. More information about formatting strings can be found on the Internet and in the **Composite Formatting** article in MSDN (http://msdn.microsoft.com/en-us/library/txafckwd.aspx). Note that the exact formatting of the output could slightly vary depending on your default culture and internationalization.

Parsing Data

The reverse operation of data formatting is data parsing. **Parsing of data (data parsing)** means to obtain a value of a given type from the text representation of this value in a specific format, i.e. **converting from text to some other data type**, the opposite of **ToString()**. For example, from the

text "10/22/2010" we can get an instance of **DateTime** type, containing the relevant date.

Often working with applications with graphical user interface requires the user input to be passed in variables of type **string**. This way we can work well with numbers and characters as well as text and dates, formatted in a user's preferred way. It is up to the developer's experience to represent the expected input data into the right way for the user. The data are then **converted to a specific data type** and processed. For example, numbers can be converted to **int** or **double** variables and then participate in mathematical expressions for calculations.

When converting types, we should not rely only on trusting the user. Always check the correctness of the input user data! Otherwise there could be an exception that could change the normal program logic.

Parsing Numeric Types

To parse a string to a number we can use the **Parse(…)** method of the primitive types. Let's see an example of parsing a string to an integer value:

```
string text = "53";
int intValue = int.Parse(text);
// intValue = 53
```

We can also parse variables of Boolean type:

```
string text = "True";
bool boolValue = bool.Parse(text);
// boolValue = true
```

Return value is **true**, when the passed parameter is initialized (not an object with **null** value), and its content is "**true**" regardless of the casing of letters in it. For example, any text such as "**true**", "**True**" or "**tRUe**" will set the variable **boolValue** to **true**. If the parameter's content is **"false"**, no matter the casing of letters, the return value will be **false**. In all other cases it throws **FormatException**.

In case the passed to the **Parse(…)** method value is invalid for the type (e.g. we pass "John!" when parsing a number), an exception is thrown.

Parsing Dates

Parsing to a date is similar to parsing to a numeric type, but it is recommended to set a specific date format. Here is an example of how this can happen:

```
string text = "11/11/2001";
DateTime parsedDate = DateTime.Parse(text);
Console.WriteLine(parsedDate);
// 11-Nov-01 0:00:00 AM
```

Whether the date will be parsed successfully and in what format exactly it will be printed on the console depends strongly on the current culture of Windows. In the example, a modified version of the U.S. culture (en-US) is used. If we want to set a format explicitly, which does not depend on the culture, we can use the method **DateTime.ParseExact(…)** and specify particular formatting pattern of our choice:

```
string text = "11/12/2001";
string format = "MM/dd/yyyy";
DateTime parsedDate = DateTime.ParseExact(
  text, format, CultureInfo.InvariantCulture);
Console.WriteLine("Day: {0}\nMonth: {1}\nYear: {2}",
  parsedDate.Day, parsedDate.Month, parsedDate.Year);
// Day: 12
// Month: 11
// Year: 2001
```

When parsing with an explicitly set format, it is required to pass a specific **culture** from which to take information about **date format** and **separators** between days and years. Since we want the parsing not to depend on a particular culture, we explicitly specify the neutral culture to be used: **CultureInfo.InvariantCulture**. To use the class **CultureInfo**, we must include the namespace **System.Globalization** in the beginning of our C# source code.

Exercises

1. **Describe the strings in C#**. What is typical for the **string** type? Explain which the most important methods of the string class are.
2. Write a program that reads a string, **reverse** it and prints it to the console. For example: "**introduction**" → "**noitcudortni**".
3. Write a program that **checks whether the parentheses are placed correctly** in an arithmetic expression. Example of expression with correctly placed brackets: **((a+b)/5-d)**. Example of an incorrect expression: **)(a+b))**.
4. How many backslashes you must specify as an argument to the method **Split(…)** in order to **split the text by a backslash**?

   Example: **one\two\three**.

Note: In C# backslash is an escaping character.

5. Write a program that detects how many times a substring is contained in the text. For example, let's look for the substring "**in**" in the text:

```
We are living in a yellow submarine. We don't have anything
else. Inside the submarine is very tight. So we are drinking
all the day. We will move out of it in 5 days.
```

The result is 9 occurrences.

6. A text is given. Write a program that **modifies the casing** of letters to uppercase at all places in the text surrounded by **<upcase>** and **</upcase>** tags. Tags cannot be nested.

Example:

```
We are living in a <upcase>yellow submarine</upcase>. We
don't have <upcase>anything</upcase> else.
```

Result:

```
We are living in a YELLOW SUBMARINE. We don't have ANYTHING
else.
```

7. Write a program that reads a string from the console (20 characters maximum) and if shorter complements it right with "*****" to 20 characters.

8. Write a program that converts a given string into the form of array of Unicode escape sequences in the format used in the C# language. Sample input: "**Test**". Result: "**\u0054\u0065\u0073\u0074**".

9. Write a program that **encrypts a text** by applying XOR (excluding or) operation between the given source characters and given cipher code. The encryption should be done by applying XOR between the first letter of the text and the first letter of the code, the second letter of the text and the second letter of the code, etc. until the last letter of the code, then goes back to the first letter of the code and the next letter of the text. Print the result as a series of Unicode escape characters **\xxxx**.

Sample source text: "**Test**". Sample cipher code: "**ab**". The result should be the following: "**\u0035\u0007\u0012\u0016**".

10. Write a program that **extracts from a text all sentences that contain a particular word**. We accept that the sentences are separated from each other by the character "**.**" and the words are separated from one another by a character which is not a letter. Sample text:

```
We are living in a yellow submarine. We don't have anything
else. Inside the submarine is very tight. So we are drinking
```

```
all the day. We will move out of it in 5 days.
```

Sample result:

```
We are living in a yellow submarine.
We will move out of it in 5 days.
```

11. A string is given, composed of several **"forbidden" words** separated by commas. Also a text is given, containing those words. Write a program that **replaces the forbidden words with asterisks**. Sample text:

```
Microsoft announced its next generation C# compiler today.
It uses advanced parser and special optimizer for the
Microsoft CLR.
```

Sample string containing the forbidden words: "**C#,CLR,Microsoft**".

Sample result:

```
********* announced its next generation ** compiler today.
It uses advanced parser and special optimizer for the
********* ***.
```

12. Write a program that reads a number from console and prints it in **15-character field, aligned right** in several ways: as a decimal number, hexadecimal number, percentage, currency and exponential (scientific) notation.

13. Write a program that **parses an URL** in following format:

```
[protocol]://[server]/[resource]
```

It should **extract** from the URL the protocol, server and resource parts. For example, when **http://www.cnn.com/video** is passed, the result is:

```
[protocol]="http"
[server]="www.cnn.com"
[resource]="/video"
```

14. Write a program that **reverses the words in a given sentence** without changing punctuation and spaces. For example: "**C# is not C++ and PHP is not Delphi**" → "**Delphi not is PHP and C++ not is C#**".

15. A dictionary is given, which consists of several lines of text. Each line consists of a **word and its explanation**, separated by a hyphen:

```
.NET – platform for applications from Microsoft
```

```
CLR – managed execution environment for .NET
namespace – hierarchical organization of classes
```

Write a program that **parses the dictionary** and then reads words from the console in a loop, **gives an explanation** for it or writes a message on the console that the word is not into the dictionary.

16. Write a program that **replaces all hyperlinks** in a HTML document consisting of **…** and hyperlinks in "forum" style, which look like **[URL=…]…[/URL]**.

    Sample text:

```
<p>Please visit <a href="http://softuni.org">our site</a> to
choose a training course. Also visit <a href=
"http://forum.softuni.org">our forum</a> to discuss the
courses.</p>
```

    Sample result:

```
<p>Please visit [URL=http://softuni.org]our site[/URL] to
choose a training course. Also visit [URL=
http://forum.softuni.org]our forum[/URL] to discuss the
courses.</p>
```

17. Write a program that **reads two dates** entered in the format "**day.month.year**" and calculates the **number of days between them**.

```
Enter the first date: 27.02.2006
Enter the second date: 3.03.2006
Distance: 4 days
```

18. Write a program that reads the date and time entered in the format "**day.month.year hour:minutes:seconds**" and prints the date and time after 6 hours and 30 minutes in the same format.

19. Write a program that **extracts all e-mail addresses** from a text. These are all substrings that are limited on both sides by text end or separator between words and match the shape **<sender>@<host>…<domain>**. Sample text:

```
Please contact us by phone (+001 222 222 222) or by email at
example@gmail.com or at test.user@yahoo.co.uk. This is not
email: test@test. This also: @gmail.com. Neither this:
a@a.b.
```

    Extracted e-mail addresses from the sample text:

```
example@gmail.com
test.user@yahoo.co.uk
```

20. Write a program that **extracts from a text all dates** written in format **DD.MM.YYYY** and prints them on the console in the standard format for Canada. Sample text:

```
I was born at 14.06.1980. My sister was born at 3.7.1984. In
5/1999 I graduated my high school. The law says (see section
7.3.12) that we are allowed to do this (section 7.4.2.9).
```

Extracted dates from the sample text:

```
14.06.1980
3.7.1984
```

21. Write a program that extracts from a text all words which are **palindromes**, such as **ABBA**", "**lamal**", "**exe**".
22. Write a program that reads a string from the console and prints in alphabetical order **all letters from the input string and how many times each one of them occurs** in the string.
23. Write a program that reads a string from the console and prints in alphabetical order **all words from the input string** and **how many times each one of them occurs** in the string.
24. Write a program that reads a string from the console and replaces every sequence of identical letters in it with a single letter (the **repeating** letter). Example: "**aaaaabbbbbcdddeeeedssaa**" → "**abcdedsa**".
25. Write a program that reads a list of words separated by commas from the console and prints them in alphabetical order (after **sorting**).
26. Write a program that **extracts all the text without any tags and attribute values** from an HTML document.

Sample text:

```
<html>
  <head><title>News</title></head>
  <body><p><a href="http://softuni.org">Software
    University</a>aims to provide free real-world practical
    training for young people who want to turn into
    skillful software engineers.</p></body>
</html>
```

Sample result:

```
News
Software University aims to provide free real-world
practical training for young people who want to turn into
skillful software engineers.
```

Solutions and Guidelines

1. Read in MSDN or refer to the start of this chapter.
2. Use **`StringBuilder`** and **`for`** (or **`foreach`**) loop.
3. Use **counting of the brackets**: For an opening bracket increase the counter by 1 and for closing bracket decrease it by 1. Watch the counter not to become a negative number and always ends with 0.
4. If you do not know how many slashes you must use, try **`Split(…)`** with an **increasing number of slashes** until you reach the desired result.
5. Reverse the casing of letters in text to small and **search the given substring in a loop**. Remember to use **`IndexOf(…)`** with a start index in order to avoid infinite loop.
6. Use **regular expressions** or **`IndexOf(…)`** method for opening and closing tag. Calculate the start and end index of the text. Change the text in all capital letters and replace the entire substring **opening tag + text + closing tag** with the text in uppercase.
7. Use the **`PadRight(…)`** method from the **`String`** class.
8. Use format string "**`\u{0:x4}`**" for the Unicode character code for each character of the input string (you can get it by converting **`char`** to **`ushort`**).
9. Let the cipher **`cipher`** consists of **`cipher.Length`** letters. Iterate through all letters in the text and encrypt the letter at position **`index`** in the text with **`cipher[index % cipher.Length]`**. If you have a letter from the text and letter from the cipher, we can perform **XOR** operation between them by transforming in advance the two letters into numbers of type **`ushort`**. We can print the result with "**`\u{0:x4}`**" format string.
10. First **split the sentences** from each other by using the **`Split(…)`** method. Then make sure that each sentence **contains the searched word** by searching for it as a substring with **`IndexOf(…)`** and if you find it check whether there is a separator (character, which is not a letter or start / end of the string) on the left and on the right of the found substring.
11. First, **split the forbidden words** with the method **`Split(…)`** in order to get them as an array. For each forbidden word, iterate through the text and **search for an occurrence**. If a forbidden word is found, replace it with as many asterisks as letters contained in the forbidden word.

Another, easier approach is to use **RegEx.Replace(…)** with a suitable regular expression and a suitable **MatchEvaluator** method.

12. Use appropriate **formatting strings**.
13. Use a **regular expression** or search for the respective splitters – two slashes for a protocol and one slash as a separator between the server and the resource. Test the special cases like **missing parts of the URL**.
14. You can solve the problem in two steps: **reverse the input string**; **reverse each word in the result string**.

    Another interesting approach is to **split the input text by punctuation marks** between words, in order to get just the words of the text and then **split by the letters** to get the punctuation marks of the text. Thus, given a list of words and a list of punctuation marks between them, you can easily reverse the words, preserving the punctuation marks.
15. You can **parse the text** by splitting it by the new line character, then a second time by the "-" character. The most appropriate way to record the dictionary is in a hash table (**Dictionary<string, string>**), which will provide a quick search for a given word. Read on the Internet for hash-tables and the **Dictionary<K,T>** class. You might also check the chapter "Dictionaries, hash-Tables and Sets".
16. Using a **regular expression** is the easiest way to solve the task.

    If you still choose not to use regular expressions, you can find all substrings that start with "**<a href=**" and end with "****" and within them to replace "****" with "**]**" and then "****" with "**[/URL]**".
17. Use the methods in the **DateTime** structure. For parsing the dates you can use splitting by "**.**" or parsing with the **DateTime.ParseExact(…)** method.
18. Use the **DateTime.ToString()** and **DateTime.ParseExact()** methods with suitable formatting strings.
19. Use **RegEx.Match(…)** with an appropriate **regular expression**.

    If you want to solve the task without regular expressions, you will need to process the text letter by letter from start to finish and process the next character, depending on the current mode, which can be one of **OutsideOfEmail**, **ProcessingSender** or **ProcessingHostOrDomain**. If a separator or the end of the text is reached and host or domain is processed (mode **ProcessingHostOrDomain**), then you have found an e-mail, otherwise potentially a new e-mail is starting and mode must be changed to ProcessingSender. If **@** character is reached in **ProcessingSender** mode, **ProcessingSender** is switched on. When meeting letters or dot in **ProcessingSender** or **ProcessingHostOrDomain** mode, they ate accumulated in a buffer. You can look at all possible

groups of characters encountered respectively in each of the three modes and process them appropriately. We come to something like a final automaton (state machine), which detects e-mail addresses. All found e-mail addresses must be checked whether they have nonempty recipient, host, and domain with a length between 2 and 4 letters, as well as not beginning or ending with a dot.

Another easier approach to this problem is to split the text by all characters that are not letters and dots and to verify that the extracted "words" are valid e-mail addresses. Check can be done through an attempt to split them to nonempty parts: **<sender>**, **<host>**, **<domain>**, meeting the listed conditions.

20. Use **RegEx.Match(…)** with an appropriate **regular expression**. Alternative option is to implement a state-machine that has several states **OutOfDate**, **ProcessingDay**, **ProcessingMonth**, **ProcessingYear** and while processing the text letter by letter to move between states according to the current letter which you are processing. As in the previous task, you can extract all "words" from the text in advance and then check which ones correspond to the date template.
21. Split the text into words and check whether each word is a **palindrome**.
22. Use an array of integers **int[65536]**, which will keep **how many times each letter occurs**. Initially, all array elements are zeros. After processing the input string letter by letter you can write in the array how many times each letter occurs. For example, if you meet the letter '**A**', the number of occurrences in the array index of 65 (Unicode code '**A**') will increase by one. Finally, all non-zero elements (convert array index to **char**, to get the letter) and their number of occurrences can be printed with one scan of the array.
23. Use a hash table (**Dictionary<string, int>**) which keeps how many times each word occurs in the input string. Read on the Internet for class **System.Collections.Generic.Dictionary<K,T>**. With iteration through words you can accumulate information for each word occurrences in the hash table and with hash table iteration you can print the result.
24. You can scan text from left to right and when the current letter is identical with the previous one, miss it, but otherwise concatenate it in **StringBuilder**.
25. Use the static method **Array.Sort(…)** after parsing the input text into array of strings.
26. **Scan the text letter by letter** and at all times keep in a variable whether currently there is an opening tag which has not been closed or not. If you have "**<**", enter in **"opening tag" mode**. If you have "**>**", exit the "opening tag" mode. If you have a letter, add it to the result only if the program is not in "opening tag". After closing a tag you can add a space in order not to "stick" the text before and after the tag.