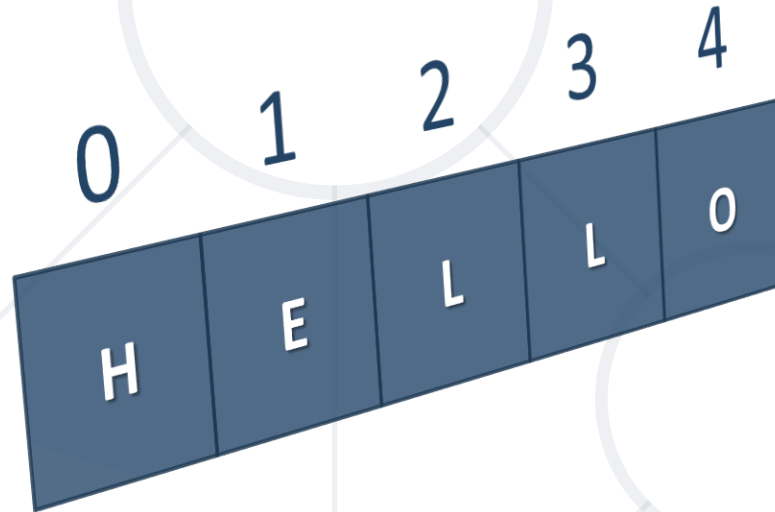


# Text Processing and Regular Expressions

Manipulating Text  
Using the .NET String Class and using RegEx



SoftUni Team  
Technical Trainers



SoftUni  
Foundation



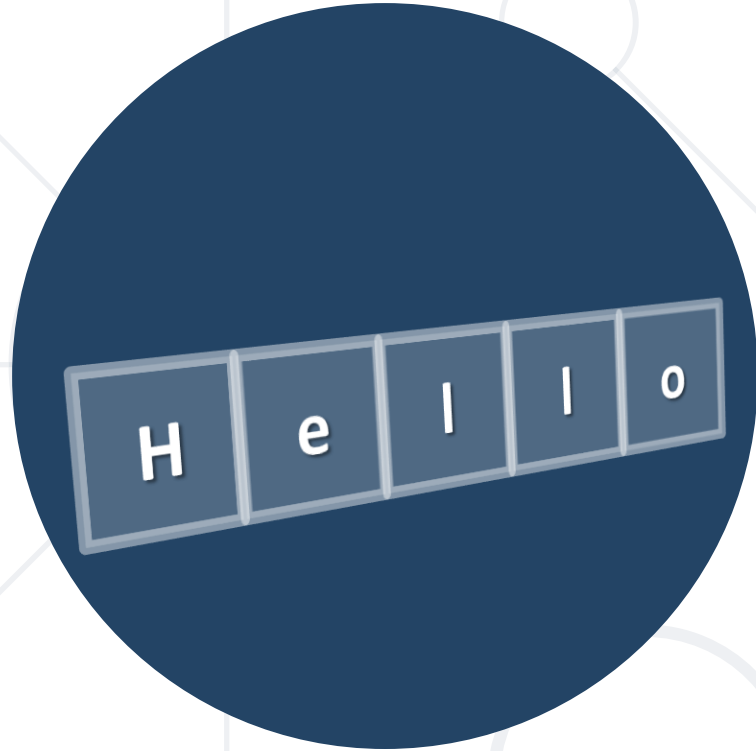
Software University

<http://softuni.bg>

# Table of Contents

1. What is a String?
2. Manipulating Strings
3. Building and Modifying Strings
4. Regular Expressions Syntax
5. Quantifiers and Grouping
6. Lookbehind/Lookahead
7. Backreferences






# Strings

## What is String?

# What is a String?

- 
- Strings are **sequences of characters** (texts)
  - The string data type in C#:
    - Is declared by the **string** keyword
    - **System.String** .NET data type
  - Strings are enclosed in quotation marks:
- ```
string s = "Hello, C#";
```
- They can be concatenated with the **+** operator:

```
string s = "Hello" + " " + "C#";
```

# In C# Strings are Immutable

- Strings are **immutable** (read-only) sequences of characters
- Accessible by index (**read-only**)

```
string str = "Hello, C#";  
char ch = str[2]; // OK  
str[2] = 'a';    // Error!
```

- Strings use **Unicode** – they can use different alphabets

```
string greeting = "你好"; // (Lí-hó) Taiwanese
```



# Initializing a String

- Initializing from a string literal:

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

- Reading a **string** from the console:

```
string name = Console.ReadLine();  
Console.WriteLine("Hi, " + name);
```

- Converting a **string** from and to a **char array**:

```
string str = new String(new char[] { 's', 't', 'r' });  
char[] charArr = str.ToCharArray();  
// ['s', 't', 'r']
```





# Manipulating Strings

- Use the **+** or the **+=** operators

```
string text = "Hello" + ", " + "world!";  
// "Hello, world!"
```

```
string text = "Hello, ";  
text += "John"; // "Hello, John"
```

- Use the **Concat()** method

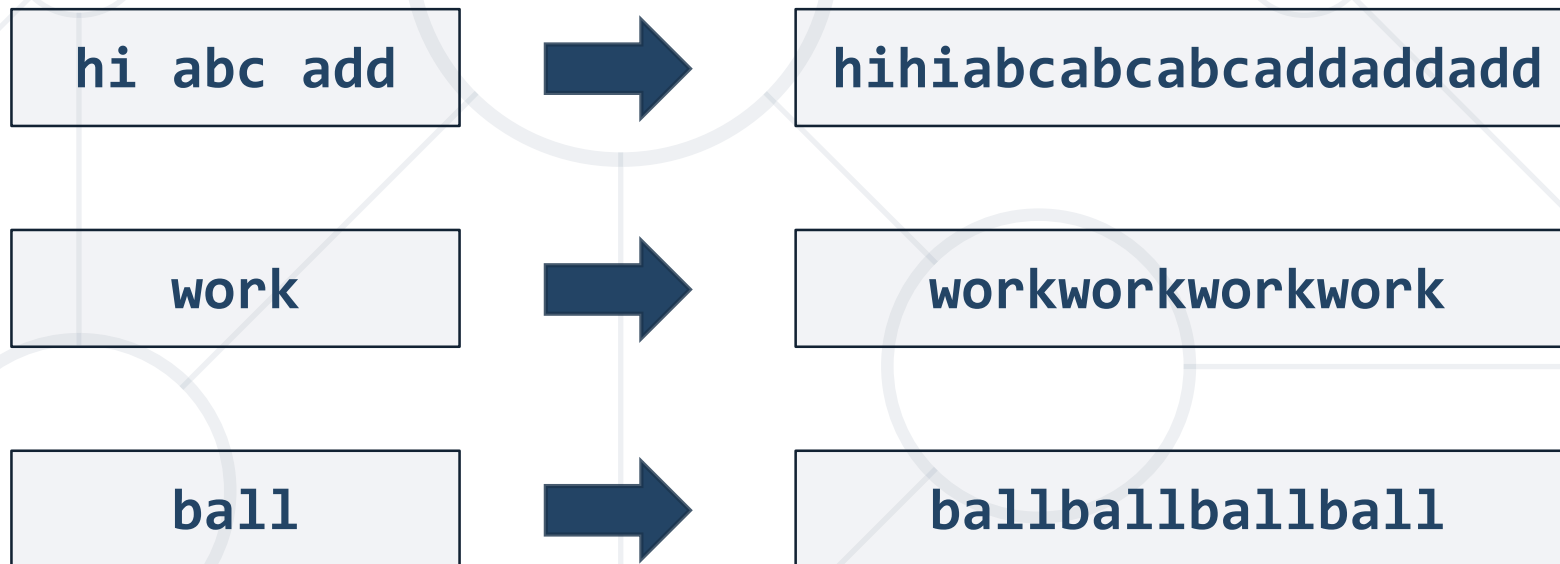
```
string greet = "Hello, ";  
string name = "John";  
string result = string.Concat(greet, name);  
Console.WriteLine(result); // "Hello, John"
```





# Problem: Repeat Strings

- Read an array from strings
- Repeat each word **n** times, where **n** is the length of the word



# Solution: Repeat Strings

```
string[] words = Console.ReadLine().Split();
string result = "";
foreach (string word in words)
{
    int repeatTimes = word.Length;
    for (int i = 0; i < repeatTimes; i++)
        result += word;
}
Console.WriteLine(result);
```

- **IndexOf()** - returns the first match index or -1

```
string fruits = "banana, apple, kiwi, banana, apple";  
Console.WriteLine(fruits.IndexOf("banana")); //0  
Console.WriteLine(fruits.IndexOf("orange")); //-1
```

- **LastIndexOf()** - finds the last occurrence

```
string fruits = "banana, apple, kiwi, banana, apple";  
Console.WriteLine(fruits.LastIndexOf("banana")); //21  
Console.WriteLine(fruits.LastIndexOf("orange")); //-1
```

- **Contains()** - checks whether one string contains other string

```
string text = "I love fruits.";
Console.WriteLine(text.Contains("fruits")); //True
Console.WriteLine(text.Contains("banana")); //False
```

- **Substring(int startIndex, int length)**

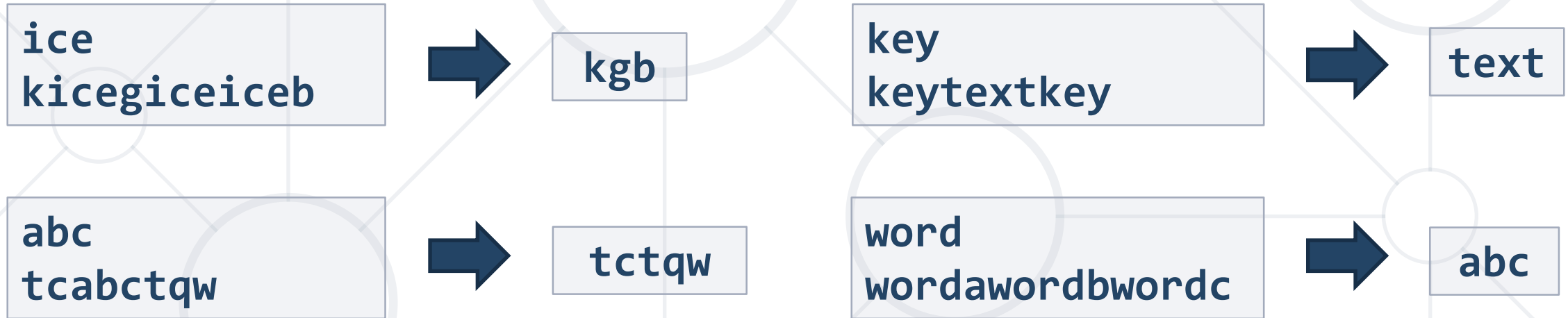
```
string card = "10C";  
string power = card.Substring(0, 2);  
Console.WriteLine(power); //10
```

- **Substring(int startIndex)**

```
string text = "My name is John";  
string extractWord = text.Substring(11);  
Console.WriteLine(extractWord); //John
```

# Problem: Substring

- You are given a **text** and a **remove word**
- Remove all substrings that are equal to the remove word



# Solution: Substring

```
string key = Console.ReadLine();  
string text = Console.ReadLine();  
  
int index = text.IndexOf(key);  
  
while (index != -1)  
{  
    text = text.Remove(index, key.Length);  
    index = text.IndexOf(key);  
}  
  
Console.WriteLine(text);
```

- **Split** a string by given **separator**

```
string text = "Hello, john@softuni.bg, you have been using  
john@softuni.bg in your registration";  
string[] words = text.Split(", ");  
  
//words[:  
//"Hello"  
//"john@softuni.bg"  
//"you have been using john@softuni.bg in your registration"
```



- **Split** can be used with multiple separators

```
char[] separators = new char[] { ' ', ',', '.', ' ' };  
string text = "Hello, I am John.";   
string[] words = text.Split(separators);  
// "Hello", "", "I", "am", "John", ""
```

- Using **StringSplitOptions.RemoveEmptyEntries** to remove the empty elements from the array

```
char[] separators = new char[] { ' ', ',', '.', ' ' };  
string text = "Hello, I am John.";   
string[] words = text  
    .Split(separators, StringSplitOptions.RemoveEmptyEntries);  
// "Hello", "I", "am", "John"
```

- **Replace(match, replacement)** – replaces all occurrences
  - The result is a new **string**

```
string text = "Hello, john@softuni.bg, you have been using john@softuni.bg in your registration.";
```

```
string replacedText = text  
    .Replace("john@softuni.bg", "john@softuni.com");
```

```
Console.WriteLine(replacedText);
```

*//Output:*

*//Hello, john@softuni.com, you have been using john@softuni.com in your registration.*

# Problem: Text Filter

- You are given a text and a string of banned words
  - Replace all banned words in the text with asterisks

Linux, Windows

It is not Linux, it is GNU/Linux. Linux is merely the kernel, while GNU adds the functionality...



It is not \*\*\*\*\*, it is GNU/\*\*\*\*\*. \*\*\*\*\* is merely the kernel, while GNU adds the functionality...

```
string[] banWords = Console.ReadLine()
    .Split(...); // TODO: add separators
string text = Console.ReadLine();
foreach (var banWord in banWords)
{
    if (text.Contains(banWord))
    {
        text = text.Replace(banWord,
            new string('*', banWord.Length));
    }
}
Console.WriteLine(text);
```

**Contains(...)** checks  
if string contains  
another string

**Replace** a word with a sequence  
of asterisks of the same length



**Live Exercises**



# **Building and Modifying Strings**

## **Using the StringBuilder Class**

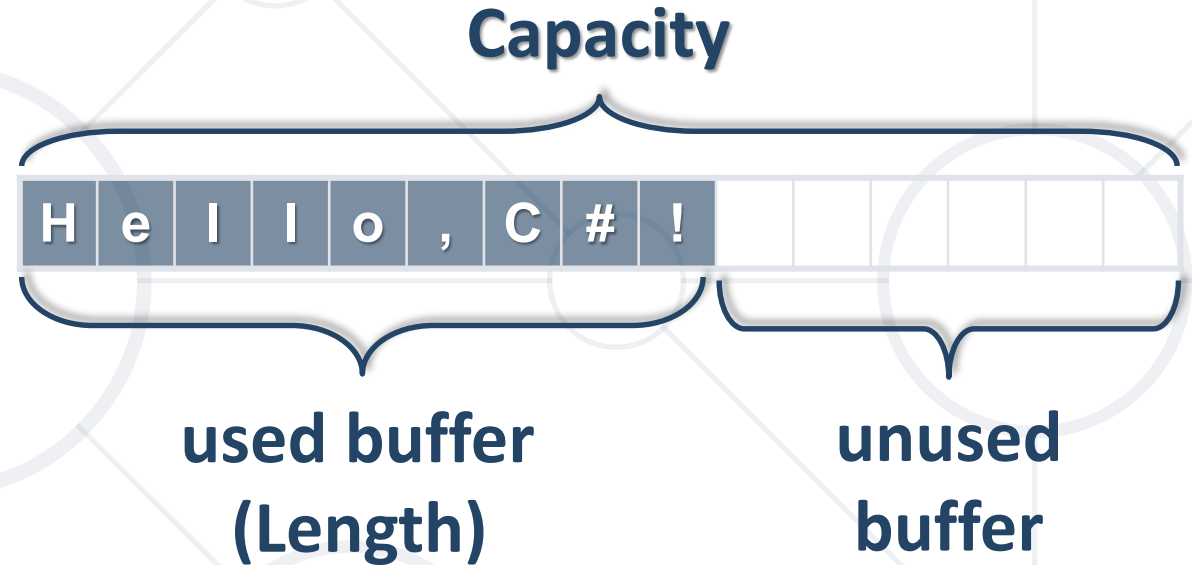
# StringBuilder: How does It Works?



**StringBuilder:**

Length = 9

Capacity = 15



- **StringBuilder** keeps a buffer space allocated in advance
  - Doesn't allocate memory for most operations → performance



- Use the **StringBuilder** to build / modify strings

```
StringBuilder sb = new StringBuilder();  
sb.Append("Hello, ");  
sb.Append("John! ");  
sb.Append("I sent you an email.");  
Console.WriteLine(sb);  
//Hello, John! I sent you an email.
```

# Concatenation vs StringBuilder (1)

- **Concatenating** strings is a **slow** operation, because every iteration **creates a new string**

```
Stopwatch sw = new Stopwatch();  
sw.Start();  
string text = "";  
for (int i = 0; i < 200000; i++)  
    text += i;  
sw.Stop();  
Console.WriteLine(sw.ElapsedMilliseconds); //73625
```



# Concatenation vs StringBuilder (2)

- Using **StringBuilder**

```
Stopwatch sw = new Stopwatch();  
sw.Start();  
StringBuilder text = new StringBuilder();  
for (int i = 0; i < 200000; i++)  
    text.Append(i);  
sw.Stop();  
Console.WriteLine(sw.ElapsedMilliseconds); //16
```



- **Append(...)** – adds a text or a string representation of an object to the end of a string

```
StringBuilder sb = new StringBuilder();  
sb.Append("Hello Peter, how are you?");
```

- **Length** – holds the length of the string in the buffer

```
sb.Append("Hello Peter, how are you?");  
Console.WriteLine(sb.Length); // 32
```

- **Clear(...)** – removes all characters

# StringBuilder Methods (2)

- **[int index]** – returns the char on the given index

```
StringBuilder sb= new StringBuilder();  
sb.Append("Hello Peter, how are you?");  
Console.WriteLine(sb[1]); // e
```

- **Insert(int index, string str)** – inserts a string at the specified character position

```
sb.Insert(11, " Ivanov");  
Console.WriteLine(sb); // Hello Peter Ivanov, how are you?
```

- **Replace(string oldValue, string newValue)** – replaces all occurrences of a specified string with another specified string

```
sb.Append("Hello Peter, how are you?");  
sb.Replace("Peter", "George");
```

- **ToString()** – converts the value of this instance to a String

```
string text = sb.ToString();  
Console.WriteLine(text);  
//Hello George, how are you?
```



**[A-Z]**

# **Regular Expressions Syntax**

## **Definition and Classes**

# What are Regular Expressions?

- Regular expressions (regex)
  - Match text by pattern
- Patterns are defined by special syntax, e.g.
  - `[0-9]+` matches a non-empty sequence of digits
  - `[A-Z][a-z]*` matches a capital + small letters
- Exercise on regex live at: [regexr.com](https://regexr.com), [regex101.com](https://regex101.com)





regular expressions 101

@regex101 donate contact bug reports & feedback wiki

REGULAR EXPRESSION 17 matches, 1035 steps (~2ms)

/ [A-Z]\w+ /g

TEST STRING SWITCH TO UNIT TESTS

Edit the Expression & Text to see matches. Roll over matches or the expression for details. Undo mistakes with ctrl-z. Save Favorites & Share expressions with friends or the Community. Explore your results with Tools. A full Reference & Help is available in the Library, or watch the video Tutorial.

Sample text for testing:

abcdefghijklmnopqrstuvwxyz ABCDEFGHIJKLMNOPQRSTUVWXYZ

0123456789 \_+-. ,!@#\$\$%^&\*();\|/|<>"'

12345 -98.7 3.141 .6180 9,000 +42

555.123.4567 +1-(800)-555-2468

[www.regex101.com](https://www.regex101.com)

Live Demo

# Regular Expression Pattern – Example

- Regular expressions (regex) describe a search pattern
- Used to find / extract / replace / split data from text by pattern

`[A-Z][a-z]+ [A-Z][a-z]+`

John Smith

Linda Davis

Contact: Alex Scott

# Character Classes: Ranges

- **[nvj]** matches any character that is either **n**, **v** or **j**

**node.js v0.12.2**

- **[^abc]** – matches any character that is **not** **a**, **b** or **c**

**Abraham**

- **[0-9]** – character range: matches any digit from **0** to **9**

**John is 8 years old.**

- `\w` – matches any **word character** (a-z, A-Z, 0-9, \_)
- `\W` – matches any **non-word character** (the opposite of `\w`)
- `\s` – matches any **white-space** character
- `\S` – matches any **non-white-space** character (opposite of `\s`)
- `\d` – matches any **decimal digit** (0-9)
- `\D` – matches any **non-decimal character** (the opposite of `\d`)



$(\w+)$

**Quantifiers and Grouping**

- \* – matches the previous element zero or more times

`\+\d*` ➔ `+359885976002 a+b`

- + – matches the previous element one or more times

`\+\d+` ➔ `+359885976002 a+b`

- ? – matches the previous element zero or one time

`\+\d?` ➔ `+359885976002 a+b`

- {3} – matches the previous element exactly 3 times

`\+\d{3}` ➔ `+359885976002 a+b`

- **(subexpression)** – captures the matched subexpression as numbered group

`\d{2}-(\w{3})-\d{4}` → `22-Jan-2015`

- **(?:subexpression)** – defines a non-capturing group

`^(?:Hi|hello),\s*(\w+)$` → `Hi, Peter`

- **(?<name>subexpression)** – defines a named capturing group

`(?<day>\d{2})-(?<month>\w{3})-(?<year>\d{4})` → `22-Jan-2015`

# Problem: Match Full Name

- Write a regular expression in [www.regex101.com](http://www.regex101.com) to match a valid full name, according to these conditions:
- A valid full name has the following characteristics:
  - It consists of **two words**.
  - Each word **starts** with a **capital letter**.
  - After the first letter, it only contains **lowercase** letters **afterwards**.
  - Each of the **two words** should be at least **two letters long**.
  - The two words are **separated** by a **single space**.



# Solution: Match Full Name

```
string regex = @"\\b[A-Z][a-z]+[\\s{1}][A-Z][a-z]+\\b";  
string names = Console.ReadLine();  
MatchCollection matched =  
    Regex.Matches(names, regex);  
  
foreach (var name in matched)  
{  
    Console.Write(name + " ");  
}
```

# Problem: Match Dates

- Write a regular expression that extracts **dates** from text
  - Valid date format: **dd-MMM-yyyy**
  - Examples: **12-Jun-1999**, **3-Nov-1999**

I am born on **30-Dec-1994**.  
My father is born on the **9-Jul-1955**.  
**01-July-2000** is not a valid date.

# Solution: Match Dates

```
string regex =  
@"\b(?<day>\d{2})([-.\\/])(?<month>[A-Z][a-z]{2})\1(?<year>\d{4})\b";  
string dates = Console.ReadLine();  
MatchCollection matched = Regex.Matches(dates, regex);  
  
foreach (Match date in matched)  
{  
    var day = date.Groups["day"].Value;  
    var month = date.Groups["month"].Value;  
    var year = date.Groups["year"].Value;  
    Console.WriteLine($"Day: {day}, Month: {month}, Year: {year}");  
}
```

# Problem: Match Phone Number

- Write a regular expression to match a **valid phone number** from **Sofia**. After you find all **valid phones**, **print** them on the console, separated by a **comma and a space** “, ”.
- A valid number has the following characteristics:
  - It starts with “**+359**”
  - Then, it is followed by the area code (always **2**)
  - After that, it’s followed by the **number** itself:
    - The number consists of **7 digits** (separated in **two groups** of **3** and **4 digits** respectively).
  - The different **parts** are **separated** by either a **space** or a **hyphen** ('-').

# Problem: Match Phone Number

```
string numbers = Console.ReadLine();
string regex = @"(?<!\d)[+]359([ -])2\1\d{3}\1\d{4}\b";

List<string> phones = new List<string>();
MatchCollection matched = Regex.Matches(numbers, regex);

foreach(Match number in matched)
{
    phones.Add(number.Value);
}
Console.WriteLine(string.Join(", ", phones));
```



? <=

**Lookahead and Lookbehind**  
Positive and negative

# Lookahead

- Positive lookahead
  - Find expression A where expression B follows

`A(?=B)`



`[a-z]+(?=\d+)`

- Negative lookahead

- Find expression A where expression B does not follow

`A(?!B)`



`[a-z]+(?!\d+)`



# Lookbehind

- Positive lookbehind
  - Find expression A where expression B precedes

`(?<=B)A` → `(?<=\d)[a-z]+`

- Negative lookbehind
  - Find expression A where expression B does not precede

`(?<!B)A` → `(?<!\d)[a-z]+`





# Backreferences Match Previous Groups

- **\number** – matches the value of a numbered capture group

```
<(\w+)[^>]*>.*?<\/\1>
```

```
<b>Regular Expressions</b> are cool!
```

```
<p>I am a paragraph</p> ... some text after
```

```
Hello, <div>I am a<code>DIV</code></div>!
```

```
<span>Hello, I am Span</span>
```

```
<a href="https://softuni.bg/">SoftUni</a>
```

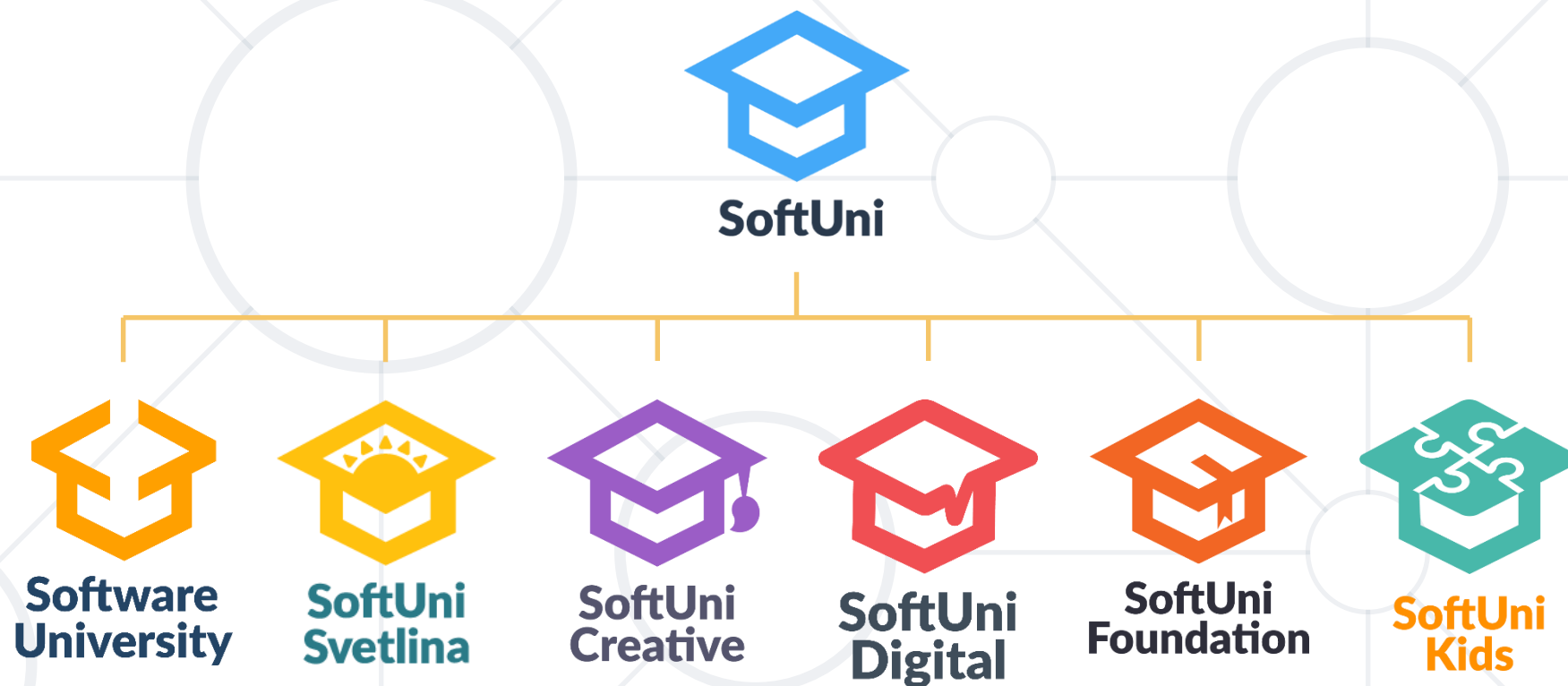


**Live Exercises**

- Strings are **immutable** sequences of Unicode characters
- String processing methods
  - **Concat()**, **IndexOf()**, **Contains()**, **Substring()**, **Split()**, **Replace()**, ...
- **StringBuilder** efficiently builds / modifies strings
- **Regular expressions** describe **patterns** for searching through text
- Can utilize **character classes**, **groups**, **quantifiers** and more



# Questions?



# SoftUni Diamond Partners



**XS**software



**SBTech**



telenor



**SoftwareGroup**  
*doing it right*

**NETPEAK**



**SmartIT**



**Postbank**

Решения за твоето утре

**SUPER  
HOSTING  
.BG**

**INDEAVR**

Serving the high achievers



**INFRAGISTICS®**

**LIEBHERR**



aeternity



# SoftUni Organizational Partners



One  
SOFTV



WORLD  
OF  
MYTHS

# Trainings @ Software University (SoftUni)

- Software University – High-Quality Education and Employment Opportunities
  - [softuni.bg](http://softuni.bg)
- Software University Foundation
  - <http://softuni.foundation/>
- Software University @ Facebook
  - [facebook.com/SoftwareUniversity](https://facebook.com/SoftwareUniversity)
- Software University Forums
  - [forum.softuni.bg](http://forum.softuni.bg)



- This course (slides, examples, demos, videos, homework, etc.) is licensed under the "Creative Commons Attribution-NonCommercial-ShareAlike 4.0 International" license

