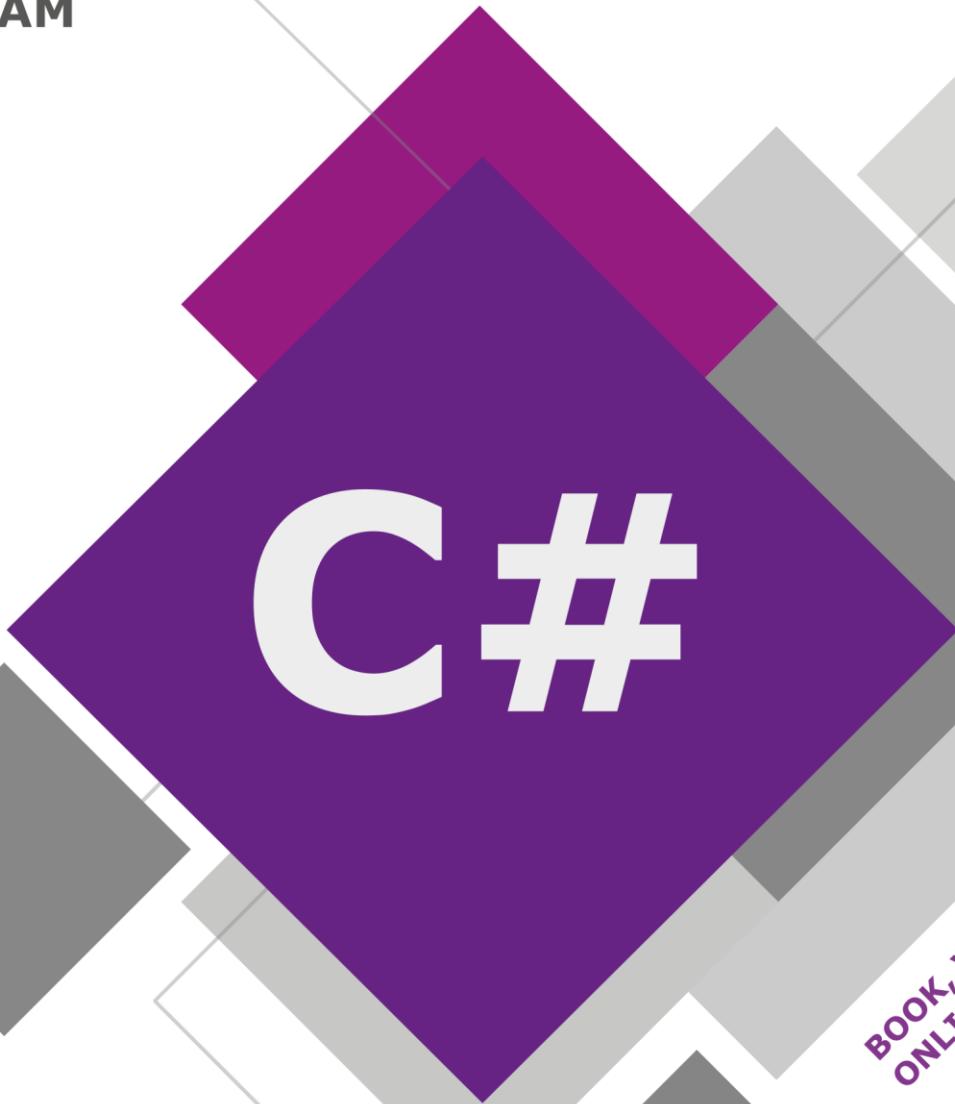


Dr. SVETLIN NAKOV

& TEAM



C#

BOOK, VIDEOS,
ONLINE JUDGE

PROGRAMMING BASICS WITH C#

Contents

Contents	3
Table of Contents	7
Preface	13
Chapter 1. First Steps in Programming	27
Chapter 2.1. Simple Calculations	61
Chapter 2.2. Simple Calculations – Exam Problems	93
Chapter 3.1. Simple Conditions	107
Chapter 3.2. Simple Conditions – Exam Problems.....	135
Chapter 4.1. More Complex Conditions	147
Chapter 4.2. More Complex Conditions – Exam Problems	171
Chapter 5.1. Loops (Repetitions).....	189
Chapter 5.2. Loops – Exam Problems	209
Chapter 6.1. Nested Loops.....	223
Chapter 6.2. Nested Loops – Exam Problems.....	243
Chapter 7.1. More Complex Loops.....	255
Chapter 7.2. More Complex Loops – Exam Problems.....	287
Chapter 8.1. Practical Exam Preparation – Part I.....	297
Chapter 8.2. Practical Exam Preparation – Part II.....	323
Chapter 9.1. Problems for Champions – Part I.....	337
Chapter 9.2. Problems for Champions – Part II	349
Chapter 10. Methods	363
Chapter 11. Tricks and Hacks	391
Conclusion	403

"Programming Basics with C#" Book and Video Lessons

The **free** book "Programming Basics with C#" introduces the readers to writing **programming code** at beginner level (variables and data, conditional statements, loops and methods) using the **C#** language. It combines **tutorial**-style learning content with **video** lessons, **code examples** and a lot of practical **coding exercises** with automated online **evaluation** (judge) system to ensure efficient learning.

Watch the promo **video** about this book and video lessons: https://youtu.be/_F606F3OgmQ.

Brief information about this edition:

- Title: Programming Basics with C#
- Authors: Svetlin Nakov & Team
- ISBN: 978-619-00-0902-3
- Edition: Faber Publishing, Sofia, 2019
- License: CC-BY-SA
- Source code: <https://github.com/SoftUni/Programming-Basics-Book-CSharp-EN>

All **authors**, **editors**, **translators** and **contributors**: Alexander Krastev, Alexander Lazarov, Angel Dimitriev, Bilyana Borislavova, Daniel Tsvetkov, Dimiter Tatarski, Dimo Dimov, Diyan Tonchev, Elena Rogleva, Hristiyan Hristov, Hristo Hristov, Iskra Nikolova, Ivelin Kirilov, Julieta Atanasova, Kalin Primov, Kalina Milanova, Karina Cholakova, Kristiyan Pamidov, Lyuboslav Lyubenov, Marieta Petrova, Marina Shideroff, Mirela Damyanova, Nelly Karaivanova, Nikolay Bankin, Nikolay Dimov, Pavlin Petkov, Peter Ivanov, Petko Dyankov, Preslav Mihaylov, Rositsa Nenova, Ruslan Philipov, Stefka Vasileva, Svetlin Nakov, Teodor Kurtev, Tonyo Zhelev, Tsvetan Iliev, Vasko Viktorov, Ventsislav Petrov, Yanitsa Valeva, Yulian Linev, Zahariya Pehlivanova, Zhivko Nedylakov.

This book is available in **several versions** in different programming languages:

- [Programming Basics with C# \(English\)](#)
- [Programming Basics with C# \(Bulgarian\)](#)
- [Programming Basics with Java \(Bulgarian\)](#)
- [Programming Basics with JavaScript \(Bulgarian\)](#)
- [Programming Basics with Python \(Bulgarian\)](#)
- [Programming Basics with C++ \(Bulgarian\)](#)

Enjoy reading and **sign up** for the **Practical Free Training Course "Programming Basics"** (<https://softuni.org>) coming together with this book, because **programming is learned by practice**, code writing and solving many, many problems, not just by reading!

“Programming Basics with C#”

Book and Video Lessons

Svetlin Nakov and Team

Alexander Krastev

Alexander Lazarov

Angel Dimitriev

Bilyana Borislavova

Daniel Tsvetkov

Dimiter Tatarski

Dimo Dimov

Diyan Tonchev

Elena Rogleva

Hristiyan Hristov

Hristo Hristov

Iskra Nikolova

Ivelin Kirilov

Julieta Atanasova

Kalin Primov

Kalina Milanova

Karina Cholakova

Kristiyan Pamidov

Lyuboslav Lyubenov

Marieta Petrova

Marina Shideroff

Mirela Damyanova

Nelly Karaivanova

Nikolay Bankin

Nikolay Dimov

Pavlin Petkov

Peter Ivanov

Petko Dyankov

Preslav Mihaylov

Rositsa Nenova

Ruslan Philipov

Stefka Vasileva

Svetlin Nakov

Teodor Kurtev

Tonyo Zhelev

Tsvetan Iliev

Vasko Viktorov

Ventsislav Petrov

Yanitsa Valeva

Yulian Linev

Zahariya Pehlivanova

Zhivko Nedylakov

ISBN: 978-619-00-0902-3

<https://csharp-book.softuni.org>

Sofia, 2019

“Programming Basics with C#” Book and Video Lessons

© Svetlin Nakov and Team, 2019

First Edition, September 2019

This book and video lessons are distributed **freely** under the [CC-BY-SA](#) open source license, which defines the following **rights** and **obligations**:

- **Sharing** – you can copy and distribute the book and videos freely in any format or media.
- **Adaptation** – you can copy, remix and modify portions of this book and video lessons and produce new content based on them.
- **Attribution** – when you use portions of this book and video lessons you should attribute the original source, along with link to it and this license, but not in any way that suggests the licensor endorses you or your use.
- **Share Alike** – if you remix, transform, or build upon this book and video lessons, you must distribute your contributions under the same license as the original.

All trademarks used in this book are the property of their respective owners.

Publisher: Faber Publishing, Veliko Tarnovo

ISBN: 978-619-00-0902-3

Official Web Site: <https://csharp-book.softuni.org>

Official Facebook Page: <https://fb.com/IntroProgrammingBooks>

Cover: Marina Shideroff – <https://behance.net/marinashideroff>

Source Code: <https://github.com/SoftUni/Programming-Basics-Book-CSharp-EN>

Table of Contents

Contents	3
Table of Contents	7
Preface	13
Video: Book + Video Course Overview	13
The Book Uses C# and Visual Studio	13
Official Textbook at SoftUni.....	13
Who Is This Book Intended for?	13
Why Did We Choose C#?	14
Learning Resources: Code + Videos + Exercises + Judge	14
Programming Is Learned by Writing, Not Reading!	14
The Software University (SoftUni).....	15
The Automated Judge System.....	16
How to Become a Software Developer?	17
More About the Book	22
Chapter 1. First Steps in Programming	27
Video: Chapter Overview.....	27
Introduction to Coding by Examples.....	27
Computer Programs – Concepts.....	28
Languages, Compilers, Interpreters and Environments	29
Runtime Environments, Low-Level and High-Level Languages.....	31
Computer Programs – Examples.....	33
Development Environments (IDE) and Visual Studio	34
Example: Creating a Console Application "Hello C#"	39
Typical Mistakes in C# Programs	43
Exercises: First Steps in Coding.....	45
Lab: Graphical and Web Applications.....	51
Chapter 2.1. Simple Calculations	61
Video: Chapter Overview.....	61
Introduction to Simple Calculations by Examples.....	61
The System Console	62
Reading Integers from the Console	62
Example: Calculating a Square Area	63
Data Types and Variables	64
Declaring and Using Variables	64
Reading Floating Point Numbers from the Console	65
Reading a Text from the Console	65
Printing and Formatting Text and Numbers.....	66
Arithmetic Operations.....	68
Concatenating Text and Numbers.....	70
Numerical Expressions.....	70

8 Programming Basics with C#

Exercises: Simple Calculations.....	72
Lab: GUI Applications with Numerical Expressions.....	87
Useful Web Sites for C# Developers.....	91
Chapter 2.2. Simple Calculations – Exam Problems.....	93
Simple Calculations – Quick Review	93
Exam Problems.....	94
Chapter 3.1. Simple Conditions.....	107
Video: Chapter Overview.....	107
Introduction to Simple Conditions by Examples	107
Comparing Numbers.....	107
Simple If Conditions	108
If-Else Conditions – Examples.....	111
Variable Scope.....	112
Sequence of If-Else Conditions	113
Debugging: Simple Operations with Debugger	114
Exercises: Simple Conditions	115
Lab: GUI (Desktop) Application – Currency Converter.....	130
Chapter 3.2. Simple Conditions – Exam Problems.....	135
Simple Conditions – Quick Review.....	135
Problem: Transportation Price	135
Problem: Pipes in Pool.....	137
Problem: Sleepy Tom Cat.....	139
Problem: Harvest.....	141
Problem: Firm	143
Chapter 4.1. More Complex Conditions	147
Video: Chapter Overview.....	147
Introduction to Complex Conditions by Examples	147
Nested If-Else Conditions	147
More Complex Conditions	150
Logical "AND"	151
Logical "OR"	153
Logical Negation (NOT)	154
More Complex Conditions – Examples	154
Switch-Case Conditional Statement.....	158
Exercises: More Complex Conditions	160
Lab: * GUI (Desktop) Application: Point and Rectangle	164
Chapter 4.2. More Complex Conditions – Exam Problems.....	171
More Complex Conditions – Quick Review	171
Problem: On Time for the Exam.....	172
Problem: Trip	175
Problem: Operations with Numbers.....	178

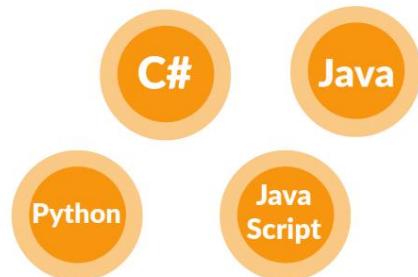
Problem: Game Tickets.....	181
Problem: Hotel Room.....	184
Chapter 5.1. Loops (Repetitions).....	189
Video: Chapter Overview.....	189
Introduction to Simple Loops by Examples	189
For Loops (Repeating Code Blocks)	190
Example: Numbers from 1 to 100	190
Example: Numbers up to 1000, Ending by 7	191
Example: All Latin Letters.....	192
Code Snippet for the for Loop in Visual Studio	192
Exercises: Loops (Repetitions).....	193
Lab: Turtle Graphics GUI Application	201
Exercises: Turtle Graphics.....	206
Chapter 5.2. Loops – Exam Problems	209
For Loops – Quick Review.....	209
Problem: Histogram	209
Problem: Smart Lilly	213
Problem: Back to the Past.....	215
Problem: Hospital.....	217
Problem: Division without Remainder	220
Problem: Logistics	221
Chapter 6.1. Nested Loops.....	223
Video: Chapter Overview.....	223
Introduction to Nested Loops by Examples	223
Nested Loops – Concepts	224
Exercises: Drawing Figures.....	229
Lab: Drawing Ratings in Web.....	236
Chapter 6.2. Nested Loops – Exam Problems.....	243
Nested Loops – Quick Review	243
Problem: Drawing a Fort	243
Problem: Butterfly.....	245
Problem: "Stop" Sign.....	247
Problem: Arrow	249
Problem: Axe	251
Chapter 7.1. More Complex Loops.....	255
Video: Chapter Overview.....	255
Introduction to More Complex Loops by Examples	255
For Loop with Step	256
While Loop.....	259
Greatest Common Divisor (GCD)	261
Do-While Loop	262

Infinite Loops with Break.....	264
Nested Loops and Break	267
Handling Errors: Try-Catch	269
Exercises: More Complex Loops	271
Lab: Web Application with Complex Loops	276
Chapter 7.2. More Complex Loops – Exam Problems.....	287
More Complex Loops – Quick Review	287
Problem: Dumb Passwords Generator	287
Problem: Magic Numbers	289
Problem: Stop Number	292
Problem: Special Numbers	294
Problem: Digits	294
Chapter 8.1. Practical Exam Preparation – Part I	297
Video: Chapter Overview.....	297
The "Programming Basics" Practical Exam.....	297
Simple Calculations – Problems.....	297
Simple Conditions – Problems	301
Complex Conditions – Problems	304
Simple Loops – Problems	308
Drawing Figures – Problems	312
Nested Loops – Problems	316
Practical Exam Preparation – Summary.....	321
Chapter 8.2. Practical Exam Preparation – Part II	323
Types of Exam Problems	323
Problem: Distance	323
Problem: Changing Tiles	326
Problem: Flowers Shop	328
Problem: Grades	330
Problem: Christmas Hat	332
Problem: Letters Combination	334
Chapter 9.1. Problems for Champions – Part I.....	337
More Complex Problems on the Studied Material	337
Problem: Crossing Sequences.....	337
Problem: Magic Dates	341
Problem: Five Special Letters	344
Chapter 9.2. Problems for Champions – Part II.....	349
More Complex Problems on the Studied Material	349
Problem: Passion Shopping Days	349
Problem: Numerical Expression	353
Problem: Bulls and Cows.....	357
Chapter 10. Methods	363

Introduction by Examples	363
What Is a "Method"?	364
Methods with Parameters.....	368
Returning a Result from a Method	372
Methods Returning Multiple Values.....	375
Method Overloading	377
Nested Methods (Local Functions).....	380
Naming Methods.....	381
Good Practices When Working with Methods	382
Exercises: Methods.....	383
Chapter 11. Tricks and Hacks	391
Code Formatting	391
Naming Code Elements	393
Shortcuts in Visual Studio	394
Code Snippets in Visual Studio.....	394
Code Debugging Techniques	398
Tricks for C# Developers	400
What We Learned in This Chapter?.....	402
Conclusion.....	403
Developer Skills	403
This Book is Only the First Step!.....	403
How to Proceed After This Book?	403
Study Software Engineering in SoftUni.....	404
Study Software Engineering in Your Own Way.....	406
Recommended Resources for Developers	406

High-quality education profession and job for software engineers

- ✓ Free start for **absolute beginners**
- ✓ Choose among the **most in-demand** professions in the software industry
- ✓ Assistance for **career start**
- ✓ **Top trainers**, mentors and active learning **community**



SoftUni provides practical, modern and **innovative education** for programming, IT, design, marketing and digital skills for thousands of young people. The "**Software University**" program builds true **software engineering professionals**.

Join the end-to-end **software engineering learning program** at SoftUni to master the most in-demand programming concepts, languages, software technologies and skills through a modern teaching methodology, **interactive learning platform** and tons of practical exercises and projects. Our **curriculum** is developed together with the **IT companies** to closely match the industry demands.

SoftUni works directly with the **companies** from the software industry to **find a job** for its students and to make them successful software engineers.

The path of the student at SoftUni



Apply now

softuni.org/apply

Preface

The book "Programming Basics with C#" introduces the readers with writing **programming code** at a beginners' level (basic coding skills), working with **development environment** (IDE), using **variables** and **data**, **operators** and **expressions**, working with the **console** (reading input data and printing output), using **conditional statements** (**if**, **if-else**, **switch-case**), **loops** (**for**, **while**, **do-while**, **foreach**) and **methods** (declaring and calling methods, passing parameters, and returning values).

Video: Book + Video Course Overview

Watch a **video lesson** about what shall we learn from this book and video course about basics of programming: <https://youtu.be/H6TWSOhav9I>.

The Book Uses C# and Visual Studio

This book and video lessons coming with it, teach **basic coding skills**, using the programming language **C#** and the development environment **Visual Studio**. All examples are given in **C#**, which is a modern, general purpose programming language, a good choice for beginners.



This book only gives you **the first steps to programming**. It covers very basic skills that you must develop for years, in order to reach a high enough level to start working as a programmer.

Official Textbook at SoftUni

This book is the official textbook for the free **Programming Basics** course for absolute beginners at the **Software University (SoftUni)** – <https://softuni.org>. The curriculum provides basic training for a deeper study of programming and prepares readers for the **entrance exam** in SoftUni.

The book is also used as unofficial **textbook for school-level programming courses** in the high schools, studying professions like "**Programmer**", "**Application Programmer**" and "**System Programmer**", as well as an additional teaching tool in the initial programming courses at the **secondary schools**, **mathematical and professional high schools**.

Who Is This Book Intended for?

That book is suitable for **absolute beginners in programming** who want to try what programming is and learn the main constructions for writing programming code that are used in software development, regardless of the programming language and the technologies used. The book gives a **solid basis** of practical skills that you can use in any future training in programming and software development.

For anyone who hadn't passed [the free course on Programming Basics in SoftUni](#), we specifically recommend to sign up for it **completely free**, because one learns programming by doing it, not by reading it! During the course you will get free access to lessons, explanations and demonstrations on site or online (such as video tutorials), **a lot of practice and code writing**, help with the task solutions after each topic, access to trainers, assistants and mentors, as well as forums and discussion groups for any questions, access to a community of thousands of people who are new in programming, and any other help that a beginner might need.

The free course for beginners in SoftUni is suitable for **school students** (of age 10+), **university students** and **workers** having any other professions, who want to gain technical knowledge and check if programming is what they like to do and understand if they would like to develop in the software development field.

A new group starts each month. The "Programming Basics" course at SoftUni is organized regularly using a few different programming languages as basis. So, just check it out! The course is **free**, and you can quit any time you like. Signing up for free on-site or online training is available via the **SoftUni application form**: <https://softuni.org>.

Why Did We Choose C#?

For this book, we choose the **C# language** because it is a **modern programming language** for high-level programming, open source, easy to learn and **suitable for beginners**. Using C# is **widespread**, with a well-developed ecosystem, numerous libraries and technology frameworks, and accordingly, it gives many **perspectives** for development. C# combines paradigms of procedural, object-oriented and functional programming in a modern way with easy syntax. In this book, we will use **C# language** and **Visual Studio** development environment, which are available for free from Microsoft.

As we will explain later, **the programming language that we start with, does not make a significant difference**, but we still need to use some programming language, and in this book we choose C# specifically. The book can also be found mirrored in other programming languages such as Java and JavaScript (see <https://csharp-book.softuni.org>).

Learning Resources: Code + Videos + Exercises + Judge

This free coding book combines **video lessons**, text and **code examples** with explanations, practical **coding exercises** with hints and guidelines, [presentation slides](#) and an automated [judge system](#) for checking your solutions.

It is **more than a book** or tutorial. It is a carefully designed **tool for learning programming** by a lot of **practical coding**, suitable for beginners with no experience.

Programming Is Learned by Writing, Not Reading!

If someone thinks they will read a book and learn to program without writing code and solving problems, this is definitely a delusion. Programming needs **a lot of practice**, with code writing every day and solving hundreds, even thousands of problems, persistently for years.

You need **to solve a lot of problems**, to make mistakes, to fix, to search for solutions and information from the Internet, to try, to experiment, to find better solutions, to get used to the code, syntax, the programming language, the development environment, to search for errors and debugging the broken code, the algorithmic thinking, breaking the problems into smaller parts, gaining experience and raising your skills every day, because when you learn to write code, this is only the **first step to the profession of the "software engineer"**. You have a lot to learn, really!

We advise the reader, as a minimum, **to try out all the examples from the book**, to play with them, to change them and test them. Even more important than the examples are **the exercises** because they develop the programmer's practical skills. This book provides nearly **150 practical coding exercises**, so it is a good foundation for developing coding and algorithmic thinking skills.

You need to **solve all the problems in the book** because programming is learned with practice! The exercises after each topic are carefully selected to cover in depth the learning material. The purpose of solving all the problems is to provide **complete set of skills for writing programming code** at a beginner's level (which is the purpose of this book). During the courses in SoftUni we purposefully **focus on practice** and problem solving, and in most courses code writing occupies over 70% of the entire course.



Solve all the exercises in the book. Otherwise you won't learn anything! Programming is learned by writing a lot of code and solving thousands of problems!

The Software University (SoftUni)

[The Software University \(SoftUni\)](#) is the largest training center for software engineers in the South-Eastern Europe. Tens of thousands of students pass through the university every year. SoftUni was founded in 2014, as a continuation of the hard work of [Dr. Svetlin Nakov](#) in training **skillful software engineering professionals** by modern high-quality practical education, that combines fundamental knowledge with modern software technologies and a lot of practice.

Video: SoftUni and SoftUni Judge

Watch a video lesson about SoftUni and SoftUni Judge here: <https://youtu.be/TDIDXnFCzoo>.

SoftUni: High-Quality Practical Tech Education

The Software University (SoftUni) provides **quality education**, **profession**, **job** and **diploma** for programmers, software engineers and IT professionals. SoftUni builds an extremely successful and strong **connection between education and industry** by collaboration with hundreds of software companies, provides job and internship of its students, creates quality professionals for the software industry and directly responds to the needs of employers via the training process.



Free Programming Courses at SoftUni

SoftUni organizes **free programming lessons for beginners**: online and physically in few locations. The purpose is to give a chance to **everyone who is interested** in programming and technologies to **try programming** and check if they are interested and if they would get seriously involved in software development. You can sign up for the free course in **programming basics** using the SoftUni application page: <https://softuni.org>.

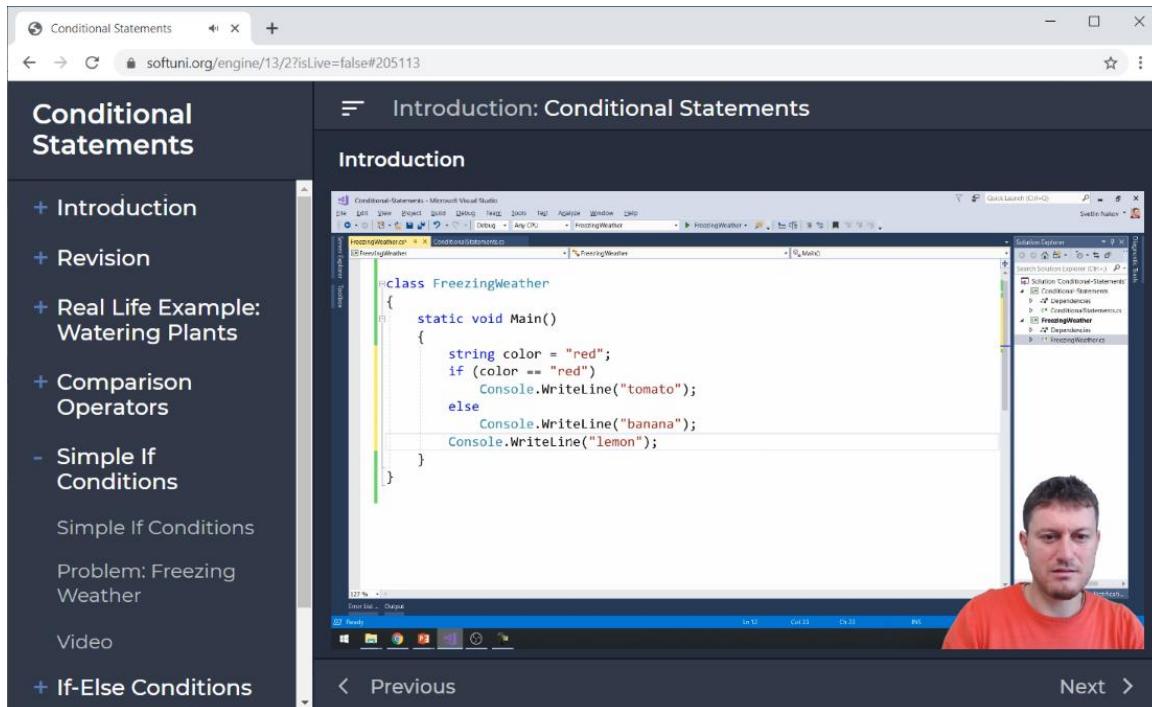
The free courses at SoftUni have the purpose to introduce you to **basic programming constructions** in the software development world, that you can use at any programming language. These include working with **data**, **variables** and **expressions**, using **conditional statements**, constructing **loops**, defining and calling **methods** and other approaches for building programming logic. The trainings are **highly practically oriented** which means that **the emphasis is strongly on exercises**, and you get the opportunity to apply your knowledge during the learning process.

This **programming book** accompanies the free programming lessons for beginners in SoftUni and serves as an additional teaching aid to help the learning process.

The SoftUni Interactive Classroom

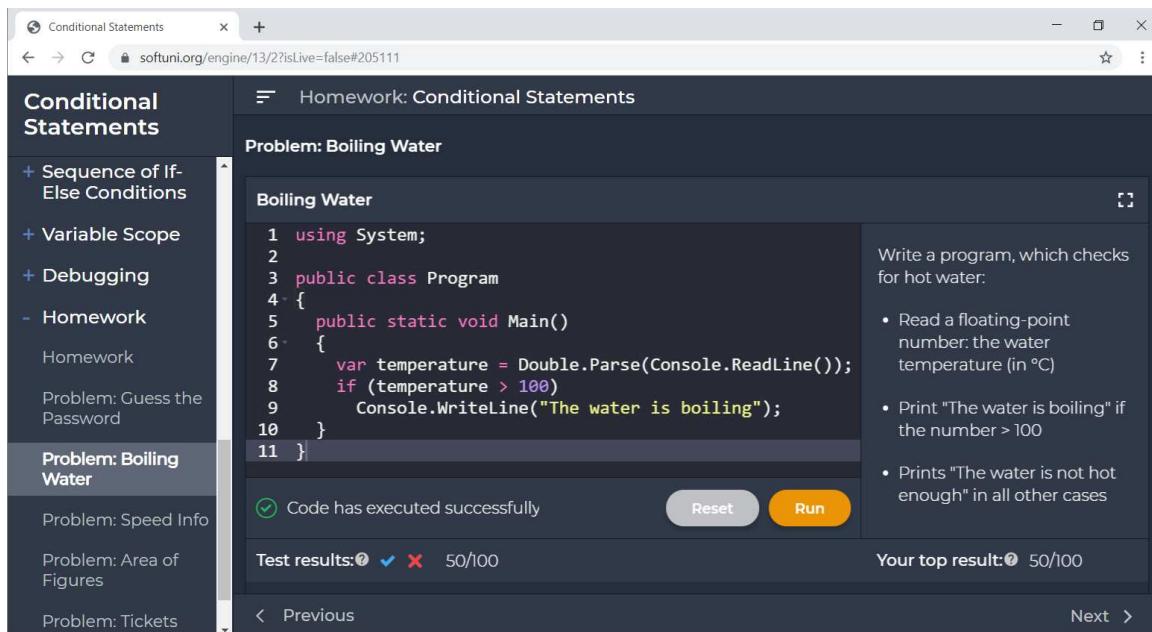
SoftUni teaches programming and trains software engineering professionals worldwide through its innovative **SoftUni Interactive Classroom** (<https://softuni.org>), which combines **video lessons** with live coding sessions, **live code** examples and interactive **live coding exercises** with live remote **real-time developer support** (live chat with the trainers), integrated into a single platform on the Web.

Using the SoftUni Interactive Classroom, you **learn directly in the Web browser**, where you **write**, **execute and test code** and your exercise solutions are automatically evaluated using an integrated **judge system**. When you have difficulties with some exercise, you **ask for help** over multiple channels: **automated hints** and guidelines and **live help** from the trainers (**live chat** with an expert from the SoftUni training team). Give it a try at: <https://softuni.org>. This is how the **SoftUni Interactive Learning Platform (Interactive Classroom)** looks like:

A screenshot of a web-based programming tutorial. On the left, there's a sidebar with a dark background containing navigation links: '+ Introduction', '+ Revision', '+ Real Life Example: Watering Plants', '+ Comparison Operators', '- Simple If Conditions' (with a 'Simple If Conditions' link below it), 'Problem: Freezing Weather', 'Video', and '+ If-Else Conditions'. The main content area has a title 'Introduction: Conditional Statements' and a sub-section 'Introduction'. It shows a Microsoft Visual Studio interface with a code editor containing the following C# code:

```
class FreezingWeather
{
    static void Main()
    {
        string color = "red";
        if (color == "red")
            Console.WriteLine("tomato");
        else
            Console.WriteLine("banana");
        Console.WriteLine("lemon");
    }
}
```

The Visual Studio interface includes a toolbar, menu bar, and a Solution Explorer window on the right showing a project named 'FreezingWeather' with files like 'FreezingWeather.cs' and 'Program.cs'.

A screenshot of a web-based programming exercise. On the left, there's a sidebar with a dark background containing navigation links: '+ Sequence of If-Else Conditions', '+ Variable Scope', '+ Debugging', '- Homework' (with a 'Homework' link below it), 'Problem: Guess the Password', 'Problem: Boiling Water' (which is highlighted in grey), 'Problem: Speed Info', 'Problem: Area of Figures', and 'Problem: Tickets'. The main content area has a title 'Homework: Conditional Statements' and a sub-section 'Problem: Boiling Water'. It shows a code editor with the following C# code:

```
1 using System;
2
3 public class Program
4 {
5     public static void Main()
6     {
7         var temperature = Double.Parse(Console.ReadLine());
8         if (temperature > 100)
9             Console.WriteLine("The water is boiling");
10    }
11 }
```

Below the code editor, there's a message 'Code has executed successfully' with a green checkmark icon, a 'Reset' button, and a 'Run' button. A 'Test results' section shows a green checkmark, a red X, and the score '50/100'. To the right, there's a problem description: 'Write a program, which checks for hot water:' followed by a list of requirements:

- Read a floating-point number: the water temperature (in °C)
- Print "The water is boiling" if the number > 100
- Prints "The water is not hot enough" in all other cases

The bottom right corner shows 'Your top result: 50/100'.

The Automated Judge System

The SoftUni Judge System (<https://judge.softuni.org>) is an automated Internet system for **checking** the solutions of programming exercises via series of tests. The submission and verification happen in **real time**: you submit the solution and within seconds you get an answer whether it is correct.

- Each **successfully** taken test gives you the points it gains.
- For a completely **correct solution** you get all the points for this problem.
- For **partially correct** solution you get part of the points for the problem.

- For a completely **wrong** solution you get 0 points.

This how the **SoftUni Judge** looks like:

The screenshot shows a web-based judge system interface. At the top, it says "Secure | https://judge.softuni.bg/Contests/Practice/Index/503#1". Below that is the title "02. Expression". The code area contains the following C# code:

```

1 using System;
2
3 class Expression
4 {
5     static void Main()
6     {
7         Console.WriteLine((3522 + 52353) * 23 - (2336*501 + 23432 - 6743) * 3);
8     }
9 }
```

Below the code, there are performance limits: "Allowed working time: 0.100 sec.", "Allowed memory: 16.00 MB", "Size limit: 16.00 KB", and "Checker: Numbers Checker". There are buttons for "C# code" and "Submit".

The "Submissions" section shows one submission with the following details:

Points	Time and memory used	Submission date
✓ 100 / 100	Memory: 7.22 MB Time: 0.015 s	14:04:00 06.06.2017

All problems from the current book are available for testing in SoftUni Judge and we strongly recommend testing them after you solve them to be sure you don't miss anything and that your solution works correctly according to the task requirements.

Keep in mind some **specifics about SoftUni Judge**:

- For each task the **judge system keeps the best score you had**. Therefore, if you upload a solution with wrong code or lower score compared to the previous one, the system won't take away your points.
- The output of your program is **compared** by the system to a strictly expected result. Every **unnecessary symbol, missing comma or space** may lead to 0 points on a particular test. The **output** that the judge expects is described in the **requirements of every task** and **nothing else should be added**.
- **Example:** If the output requires to print a number (ex. 25), do not display any descriptive messages such as **The result is: 25**, just print as it is required, i.e. only the number.

The SoftUni judge system is **available any time** via its website: <https://judge.softuni.org>.

- To sign in use your authentication for the SoftUni website: <https://softuni.org>.
- Using the SoftUni system is **free** and it's not bound with the participation in SoftUni's courses.

We are convinced that after sending a few tasks **you will like getting instant feedback** for your solutions and the Judge system will become your favorite assistant in your programming practice.

How to Become a Software Developer?

Dear readers, probably many of you have the ambition to become programmers and develop software for a living, or work in the IT area. That's why we have prepared for you a **short guide on "How to become a programmer"**, so we can help you take the first steps towards this desired profession.

You can become a programmer (working in a software company) after **at least 1-2 years of intensive training and coding every day**, solving thousands of programming tasks, development of several more serious practical projects and gaining a lot of experience with code writing and software development. You can't become a programmer for a month or two! The profession of software engineer requires **a lot of knowledge**, covered with **a lot of practice**.

Video: Become a Software Engineer – 4 Essential Skills

Watch a video lesson about the 4 essential skills, that all software engineers should have in order to be experts their profession: <https://youtu.be/qO1ckspCaxs>.

The 4 Essential Skills of the Software Developers

There are **4 main skill groups** where all programmers must have. Most of these skills are resistant in time and are not influenced by the development in specific technologies (that are changing constantly). These are the skills that **any good programmer** has and to which every beginner must strive:

- **coding** (20%)
- **algorithmic thinking** (30%)
- **computer science and software engineering concepts** (25%)
- **languages and software technologies** (25%)

Skill #1 – Coding (20%)

Writing code forms around 20% of the minimum knowledge and skills of a programmer, needed for starting a job in a software company. The skill of coding includes the following components:

- Working with variables, conditional statements, loops
- Using functions, methods, classes and objects
- Data manipulation: arrays, lists, hash tables, strings

The skill of coding **can be acquired in a few months** of hard learning and solving practical problems by writing code every day. This book covers only the first point of coding: **working with variables, conditional statements and loops**. The rest remains to be learned in follow-up trainings, courses, books and practical work on projects.

The book (and the courses based on it) gives only the beginning of one long and serious training on the way to professional programming. If you don't learn perfectly the material from this book, you can't become a programmer. You are going to miss fundamentals and it will be harder in the future. For this reason, **put enough time to programming basics**: solve many problems and write code every day for months until you learn to **solve every problem from the book very easily**. Then go ahead.

We specifically note that **the programming language does not matter** for the ability to code. You can code or not. If you can code with C#, you will easily learn to code with Java, C++ or other languages. That's why **the coding skills** are being studied quite seriously at the beginners courses in SoftUni ([curriculum](#)), and each programming book for beginners starts with them, including this one.

Skill #2 – Algorithmic Thinking (30%)

The algorithmic (logical, engineering, mathematical, abstract) thinking forms around 30% of the minimum skills of a programmer needed to start the profession. **Algorithmic thinking** is the ability to break a particular problem into a logical sequence (algorithm), to find a solution for every separate step and then assemble the steps into a working solution. That is the most important skill of any programmer. How to **develop algorithmic thinking**?

- Algorithmic thinking is developed by solving **multiple programming (1000+)** problems, as diverse as possible. That is the recipe: solving thousands of practical problems, building algorithms and implementing the algorithms, along with debugging the issues that come up in the process.
- Sciences like physics, mathematics and identical ones helps a lot, but they are not mandatory! People with **engineering and technical skills** usually learn very easily to think logically, because they already **have problem solving skills**, even if it is not algorithmic.
- The ability of **solving programming problems** (for which algorithmic thinking is needed) is extremely important for a programmer. Many companies test particularly this skill during their job interviews.

The current book develops the **beginner level of algorithmic thinking**, but it's not enough to make you a good programmer. To become good at this profession you must add **logical thinking and problem-solving skills**, beyond the range of this book. For example, working with **data structures** (arrays, lists, matrices, hash-tables, binary trees) and **basic algorithms** (searching, sorting, searching in tree structures, recursion, etc.).

Algorithmic thinking skills can be seriously developed at the beginner courses for software engineers in SoftUni (see [curriculum](#)), as well as in specialized algorithm-oriented trainings like [data structures](#) and [algorithms](#).

As you can guess **the programming language doesn't matter** for the development of algorithmic thinking. To think logically is a universal skill, and it's not related only to programming. Because of the well-developed logical thinking it's believed that programmers are smart people, and that a stupid person can't become a programmer.

Skill #3 – Computer Science and Software Engineering (25%)

Fundamental knowledge and skills for programming, software development, software engineering and computer science comprise around 25% of the developer's minimum skills to start a job. Here are the more important of these skills and knowledge:

- **Basic mathematical concepts** related to programming – coordinate systems, vectors and matrices, discrete and non-discrete mathematical functions, state automata and state machines, combinatorics and statistics concepts, algorithm complexity, mathematical modeling and others
- **Programming skills** – code writing, working with data, using conditional statements and loops, working with arrays, lists and associative arrays, strings and text processing, working with streams and files, using programming interfaces (APIs), working with IDE, debugger, developer tools, etc.
- **Data structures and algorithms** – lists, trees, hash-tables, graphs, search, sorting, recursion, binary search trees, etc.
- **Object-oriented programming (OOP)** – working with classes, objects, inheritance, polymorphism, abstraction, interfaces, data encapsulation, exceptions management, design pattern
- **Functional programming (FP)** – working with lambda functions, higher order functions, functions that return a function as a result, closure, etc.
- **Databases** – relational and non-relational databases, database modeling (tables and links between them), SQL query language, object-relational mapping (ORM), transactions and transaction management
- **Network programming** – network protocols, network communication, TCP/IP, concepts, tools and technologies from computer networks
- **Client-server** interaction, peer to peer communication, back-end technologies, front-end technologies, MVC architectures

- **Technologies for server development (back-end)** – Web server architecture, HTTP protocol, MVC architecture, REST architecture, web development frameworks, templating engines
- **Front-end technologies (client-side development)** – HTML, CSS, JS, HTTP, DOM, AJAX, communication with backend, calling REST API, front-end frameworks, basic design and UX (user experience) concepts
- **Mobile technologies** – mobile apps, Android and iOS development, mobile user interface (UI), calling server logic
- **Embedded systems** – microcontrollers, digital and analog input and output control, sensor access, peripheral management
- **Operating systems** – working with operating systems (Linux, Windows, etc.), installation, configuration and basic system administration, process management, memory, file system, users, multitasking, virtualization and containers
- **Parallel and asynchronous programming** – thread management, asynchronous tasks, promises, common resources, and access synchronization
- **Software engineering** – source control systems, development management, task planning and management, software development methodologies, software requirements and prototypes, software design, software architectures, software documentation
- **Software testing** – unit testing, test-driven development, QA engineering, error reporting and error tracking, automation testing, build processes and continuous integration

We need to clarify once again that **the programming language does not matter** for the assimilation of all these skills. They accumulate slowly, over many years of practice in the profession. Some knowledge is fundamental and can be learned theoretically, but for their full understanding and in-depth awareness, you need years of practice.

Fundamental knowledge and skills for programming, software development, software engineering, and computer science are taught in deep details during the [Software Engineering Program](#) at SoftUni (<https://softuni.org/curriculum>) as well as in the [elective courses](#). Working with a variety of software libraries, APIs, frameworks and software technologies and their interaction gradually builds this knowledge and skills, so do not expect that you will understand them from a single course, book or project.

To start working as a programmer, only **basic knowledge in the areas listed above** is enough and the improvement happens at the workplace according to the technology and development tools used in the company and the team.

Skill #4 – Programming Languages and Technologies (25%)

Programming languages and software development technologies form around 25% of a programmer's skills. They have the largest learning content, but they change very fast over time. If we look at the **job offers** in the software industry, they usually mention words like the ones below, but in the job offers they secretly mention **the three main skills**: coding, algorithmic thinking and knowing the fundamentals of computer science and software engineering.

For those clearly technological skills **the programming language does matter**.

- **Note:** only for these 25% of the profession the programming language does matter!
- **For the rest 75% of the skills the programming language doesn't matter**, and these skills are resistant in time and transportable between different languages and technologies.

Here are some commonly used software development stacks which software companies are looking for (as of September 2019):

- **C#** + OOP + FP + classes from .NET + SQL Server databases + Entity Framework (EF) + ASP.NET MVC + HTTP + HTML + CSS + JS + DOM + jQuery + cloud + containers
- **JavaScript (JS)** + OOP + FP + databases + MongoDB or MySQL + HTTP + web programming + HTML + CSS + JS + DOM + jQuery + Node.js + Express + Angular or React + cloud + containers
- **Python** + OOP + FP + databases + MongoDB or MySQL + HTTP + web development + HTML + CSS + JS + DOM + jQuery + Django or Flask + cloud + containers
- **Java** + Java API classes + OOP + FP + databases + MySQL + HTTP + web programming + HTML + CSS + JS + DOM + jQuery + JSP / Servlets + Spring MVC or Java EE / JSF + cloud + containers
- **PHP** + OOP + databases + MySQL + HTTP + web development + HTML + CSS + JS + DOM + jQuery + Laravel or Symfony or other MVC framework for PHP + cloud + containers
- **C++** + OOP + STL + Boost + native development + databases + HTTP + other languages and technologies
- **Swift** + OOP + MacOS + iOS + Cocoa + Cocoa Touch + XCode + HTTP + REST + other languages and technologies
- **Go** + OOP + Linux + Protobuf + gRPC + cloud + containers + other languages and technologies

If the words above look scary and absolutely incomprehensible, then you are at the very beginning of your career and you need **many years of learning** until you reach the profession of a "software engineer". Do not worry, every programmer goes through one or several technology stacks and needs to study **a set of interconnected technologies**, but the bottom line is **the ability to write programming logic (coding)**, and the skill of **algorithmic thinking** (to solve programming problems). It's not possible without them!

The Programming Language Doesn't Matter!

As it already became clear, **the difference between programming languages** and more specifically between the skills of developers in different languages and technologies forms around **10-20% of the developer's skills**.

- All programmers have around **80-90% of the same skills** that do not depend on the programming language! These are the skills to program and to design and develop software, that are very similar in different programming languages and development technologies.
- The more languages and technologies you know, the faster you will learn new ones, and the less you will feel the difference between them.

Indeed, **the programming language almost does not matter**, you just have to learn to program, and this starts with **coding** (this book) goes on in the more complex **programming concepts** (like data structures, algorithms, OOP and FP) and includes the use of **fundamental knowledge and skills for software development, software engineering and computer science**.

Only when you start working with a specific technology into a software project you will need **a specific programming language**, knowledge about specific programming libraries (APIs), frameworks and software technologies (front-end UI technologies, back-end technologies, ORM technologies, etc.). Keep calm, you will learn them, all programmers are learning them, but first you need to learn the foundation: **to program and do it well**.

This book uses the C# language, but it is not required and can be replaced with Java, JavaScript, Python, PHP, C++, Ruby, Swift, Go, Kotlin, or any other language. To be a **software developer**, you need to learn **coding (20%)**, learn **algorithmic thinking**, and **solve problems (30%)**, to have **fundamental knowledge of programming and computer science (25%)** and to master a **specific programming language and the technologies around it (25%)**. Be patient, for a year or two all this can be mastered on a good starting level, if you are serious.

More About the Book

The book "Programming Basics with C#" has a long story behind, involving more than 40 contributors, developers and trainers, who share their knowledge and skills to **teach the newbies generation of developers how to program.**

Video: Book Authors and Contributors

Watch a video lesson about the trainers and contributors who developed this book and video course: https://youtu.be/z_6l_mU0tv0.

The Story of This Book

The head of the project for creating the **free open-source programming book for beginners** is [Dr. Svetlin Nakov](#) (<https://nakov.com>). He is the main ideologist and author of the learning content of the free training [course "Programming Basics" in SoftUni](#), which was used as the basis of the book.

Everything started with the mass **free basic programming courses** that have been conducted in the whole country since 2014, when the "SoftUni" initiative was launched. At the beginning these courses had larger range and covered more theory, but in 2016 Dr. Svetlin Nakov completely revised, updated and simplified the whole method of teaching, strongly emphasizing on practice. This is how the core of the **learning content of this book was created.**

The free courses at SoftUni for a start in programming, are probably the most massive trainings ever conducted in South-Eastern Europe. Until 2019 the course in programming basics **was held over 200 times, in around 40 Bulgarian towns and cities** in person and multiple times online with over 100 000 participants. It was completely natural to write a **book** for the tens of thousands of participants at the SoftUni basic programming course.

Following the principle of **free software and free knowledge**, Svetlin Nakov led a team of volunteers and started this open-source project. At first the idea was to create a **free book** with the basics of programming with the C# programming language and then other similar books with other programming languages (like Java, JavaScript and Python).

The book was initially written in **Bulgarian** language in 2017 and translated into **English** in 2019.

The project is part of the passion of the [Software University Foundation](#) (<https://softuni.foundation>) to create and distribute an open learning content to teach software engineers and IT professionals.

Authors Team

This book is developed by a broad author's team of **volunteers** who dedicated their time to give away the systematized knowledge and guide you at the start of programming. Below is a list of all authors and editors (in alphabetical order):

Aleksander Krastev, Aleksander Lazarov, Angel Dimitriev, Vasko Viktorov, Ventsislav Petrov, Daniel Tsvetkov, Dimitar Tatarski, Dimo Dimov, Diyan Tonchev, Elena Rogleva, Zhivko Nedyalkov, Julieta Atanasova, Zahariya Pehlivanova, Ivelin Kirilov, Iskra Nikolova, Kalin Primov, Kristiyan Pamidov, Luboslav Lubenov, Nikolay Bankin, Nikolay Dimov, Pavlin Petkov, Petar Ivanov, Preslav Mihaylov, Rositsa Nenova, Ruslan Filipov, Svetlin Nakov, Stefka Vasileva, Teodor Kurtev, Tonyo Zhelev, Hristian Hristov, Hristo Hristov, Tsvetan Iliev, Yulian Linev, Yanitsa Vuleva.

Book cover design: **Marina Shideroff**.

Dr. Svetlin Nakov – The Leading Author

The entire project for creating this book, videos and teaching curriculum for beginners in programming is driven by **Svetlin Nakov** who inspired the other contributors to join this project and share their knowledge and skills.

Dr. **Svetlin Nakov** (<https://nakov.com>) is a passionate **software engineer**, inspirational **technical trainer** and tech **entrepreneur** from Bulgaria, experienced in broad range of programming languages, software technologies and development platforms. He is co-founder of several highly successful **tech startups** and community and non-profit organizations. Svetlin is training, innovation and inspiration leader at **SoftUni** – the largest tech education provider in South-Eastern Europe.



Svetlin Nakov has 20+ years of technical background as software engineer, software project manager, consultant, **trainer** and **entrepreneur** with rich experience with .NET, Java EE, information systems, databases, cryptography and software security, Web development, JavaScript, PHP, Python and software engineering. He is the leading **author of 15 books** on computer programming, software technologies, cryptography, C#, Java, JavaScript, Python and tens of technical and scientific publications. He is a big fan of **knowledge sharing** and is proud Wikipedia contributor, free books author and open-source supporter.

Svetlin has been a **speaker** at hundreds of conferences, seminars, meetups, courses and other trainings in the United States, Singapore, Germany, Egypt, Bulgaria and other locations. He holds a **PhD** degree in computer science (for his research on computational linguistics and machine learning), several **medals** from the **International Informatics Olympiads** (IOI) and the Bulgarian **President's award "John Atanasoff"**. He has been a part-time lecturer / trainer in Sofia University (Bulgaria), New Bulgarian University (Bulgaria), the Technical University of Sofia (Bulgaria), Ngee Ann Polytechnic (Singapore), Kingsland University (USA) and few others.

Currently **Svetlin Nakov** together with his partners drive the global expansion of the largest training center for software engineers in the South-Eastern Europe and the region – the **Software University**, where he inspires and **teaches hundreds of thousands of young people** in computer science, software development, information technologies and digital skills, and gives them a profession and a job.

Translators Team

- English translation lead and editor: [Nelly Karaivanova](#).
- Translators: Bilyana Borislavova, Kalina Milanova, Karina Cholakova, Marieta Petrova, Mirela Damyanova, Petko Dyankov.

Video Lessons Team

The video lessons for this book were recorded by:

- Svetlin Nakov
- Preslav Mihaylov

Official Book Web Site

The book **Programming Basics with C#** is available for free at the following web address:

<https://csharp-book.softuni.org>

This is the **official book site** and any new version will be uploaded there. The book is mirrored in the other programming languages mentioned on the website.

The Book in Other Languages: Java, JavaScript, Python, C++

This book on programming for beginners is available in several programming languages (or is in the process of being adapted for them):

- [Programming Basics with C# \(English\)](#)
- [Programming Basics with C# \(Bulgarian\)](#)
- [Programming Basics with Java \(Bulgarian\)](#)
- [Programming Basics with JavaScript \(Bulgarian\)](#)
- [Programming Basics with Python \(Bulgarian\)](#)
- [Programming Basics with C++ \(Bulgarian\)](#)

If you prefer a different language, select it from the list above.

Udemy Course "Comprehensive Introduction to Programming in C#"

To reach more readers, the book authors recorded a 22-hours **free video course in Udemy**, which achieved tens of thousands of enrollments from 140+ countries:

[Udemy: Comprehensive Introduction to Programming with C#](#)

This book is an **official textbook** for this Udemy course and serves as detailed tutorial, extending the course content with additional topics and more detailed explanations.

License and Distribution

The book is distributed **freely** in electronic format under an open license [CC-BY-SA](#).

The book is published and distributed **on paper** by SoftUni and you can buy a hard copy from online bookstores like Amazon.

The **source code** of the book can be found in GitHub: <https://github.com/SoftUni/Programming-Basics-Book-CSharp-EN>.

International Standard Book Number (ISBN): **978-619-00-0902-3**.

Official Facebook Page of the Book

The book has an **official Facebook page** where you can track the news about the book series "Programming Basics", new releases, events and initiatives:

fb.com/IntroProgrammingBooks

Discussion Forum for Your Questions

Ask your questions about basic programming book at the **SoftUni's FB Page** or in the official **SoftUni discussion forum**:

<https://fb.com/softuni.org>

<http://forum.softuni.org>

In these discussion channels you will get **proper response to any question associated with the content of this book** also any other questions about programming. The SoftUni community is so big that you will get a response **within a few minutes**. The trainers, assistants and mentors at SoftUni also responds on your questions. Note that the forum can hold questions in different languages (English, Bulgarian and others), but if you ask in English, you will get an answer in English.

Because of the big number of learners, you can find in the forum practically any **solution of any exercise, shared by your colleague**. Thousands of students already got the answer on the same exercise, so if you are late you can check the forum. Even though the exercises in "Programming Basics" are changing at some point, the sharing at SoftUni is always welcome and encouraged, that's why you will easily find solutions and guidance in any exercise.

If you do have a specific question, for example if you spend many hours on certain piece of code and it doesn't work correctly, you can always **ask in the forum**. You will be surprised how friendly are the SoftUni's forum participants.

Reporting Bugs

If you find **defects**, inaccuracies or bugs in the book, you can report them in the official bug tracker of the project:

<https://github.com/SoftUni/Programming-Basics-Book-CSharp-EN/issues>

We do not promise to fix everything you send us, but we do want to **always improve the quality** of the book, so the reported errors and reasonable suggestions will be reviewed.

High-quality education profession and job for software engineers

- ✓ Free start for **absolute beginners**
- ✓ Choose among the **most in-demand** professions in the software industry
- ✓ Assistance for **career start**
- ✓ **Top trainers**, mentors and active learning **community**



SoftUni provides practical, modern and **innovative education** for programming, IT, design, marketing and digital skills for thousands of young people. The "**Software University**" program builds true **software engineering professionals**.

Join the end-to-end **software engineering learning program** at SoftUni to master the most in-demand programming concepts, languages, software technologies and skills through a modern teaching methodology, **interactive learning platform** and tons of practical exercises and projects. Our **curriculum** is developed together with the **IT companies** to closely match the industry demands.

SoftUni works directly with the **companies** from the software industry to **find a job** for its students and to make them successful software engineers.

The path of the student at SoftUni



Apply now

softuni.org/apply

Chapter 1. First Steps in Programming

In this chapter, we are going to find out **what programming is** in its core and how to write simple programs and apps.

- We will get familiar with the idea of **programming languages** and development platforms, along with concepts like **compilation** and code **execution**.
- We are going to look at the **environments for software development** (IDEs) and how to work with them, in particular with **Visual Studio**.
- We will write and execute our **first program** with the programming language **C#** in **Visual Studio**.
- We will **exercise** with a couple of tasks: we will create **console-based** programs, a graphical application (**GUI**) and a **Web** application.
- We will learn how to check the correctness of the solutions from this book in **the Judge system of SoftUni**.
- We will get to know some of the **typical mistakes**, which are often made during code writing and how to prevent doing them.

Video: Chapter Overview

Watch a video about what we shall learn in this chapter here: <https://youtu.be/6RVKMIxTtg4>.

Introduction to Coding by Examples

Coding means to write **commands** for the computer, e.g.

```
Console.WriteLine("Welcome to coding");
```

Run the above code example: <https://repl.it/@nakov/welcome-to-coding-csharp>.

When **executed**, the above command prints the following text:

```
Welcome to coding
```

Several commands can be written as a sequence, called "**computer program**":

```
var size = 5;
Console.WriteLine("Size = " + size);
Console.WriteLine("Area = " + size * size);
```

Run the above code example: <https://repl.it/@nakov/square-area-csharp>.

The **result** (output) from the above program is as follows:

```
Size = 5
Area = 25
```

The above **program** (sequence of commands) consists of 3 commands:

1. Defines a variable **size** and stores an integer value **5** in it.
2. Prints the value of the variable **a**, along with some text.
3. Calculates and prints the value of the expression **a * a**.

Let's explain in greater detail what is **programming**, what is **programing language**, how to write **commands** and simple **programs** in the **C#** language, using the Visual Studio integrated development environment.

Computer Programs – Concepts

Let's start with the **concepts of computer programming**: computer programs, algorithms, programming languages, code compilation and execution.

Video: Computer Programs, Compilers, Interpreters

Watch a video lesson about the concepts of programming, programs, compilers and interpreters here: <https://youtu.be/U16C61p6m1k>.

What It Means "To Program"?

To program means to **give commands** to the computer, for example "to play a sound", "to print something on the screen" or "to multiply two numbers". When the commands are one after another, they are called **a computer program**. The text of computer programs is called **a program code** (or **a source code**, or even shorter – **code**).

Example of **command** for the computer:

```
Console.WriteLine("Welcome to coding");
```

Run the above code example: <https://repl.it/@nakov/welcome-to-coding-csharp>.

When **executed**, the above command prints the following text:

```
Welcome to coding
```

Computer Programs

Computer programs represent a sequence of commands that are written in certain **programming language**, like C#, Java, JavaScript, Python, C++, PHP, C, Ruby, Swift, Go or another. Example of computer program in C#:

```
using System;

class SquareArea
{
    public static void Main()
    {
        var size = 5;
        Console.WriteLine("Size = " + size);
        Console.WriteLine("Area = " + size * size);
    }
}
```

Run the above code example: <https://repl.it/@nakov/square-area-csharp>.

The above program defines a **class SquareArea**, holding a **method Main()**, which holds a sequence of **3 commands**:

1. Declaring and assigning a **variable**: `var size = 5;`

2. Calculating and **printing** an expression: `Console.WriteLine("Size = " + size);`
3. Calculating and **printing** an expression: `Console.WriteLine("Area = " + size * size);`

The **result** (output) from the above program is as follows:

```
Size = 5
Area = 25
```

We shall **explain in detail how to write programs in C#**, why we need to define a **class** and why we need to define a **method Main()** a bit later. Now, assume that the C# language requires all the above code in order to execute a sequence of command.

In order to write commands, we should know **the syntax and the semantics of the language** which we are working with, in our case – **C#**. Therefore, we are going to get familiar with the syntax and the semantics of the language C#, and with programming generally, in the current book, by learning step by step code writing from the simpler to the more complex programming constructions.

Algorithms

Computer programs usually execute some algorithm. **Algorithms** are a **sequence of steps**, necessary for the completion of a certain task and for gaining some expected result, something like a "recipe".

For example, if we fry eggs, we follow some recipe (an algorithm): we warm up the oil in a pan, break the eggs inside it, wait for them to fry and move them away from the stove.

Similarly, in programming **the computer programs execute algorithms**: a sequence of commands, necessary for the completion of a certain task. For example, to arrange a sequence of numbers in an ascending order, an algorithm is needed, e.g. find the smallest number and print it, then find the smallest number among the rest of the numbers and print it, and this is repeated until there are no more numbers left.

For convenience when creating programs, for writing programming code, for execution of programs and other operations related to programming, we need a **development environment**, for example Visual Studio.

Languages, Compilers, Interpreters and Environments

Let's review some concepts from computer programming: programming languages, compilers, interpreters and development environments (IDEs).

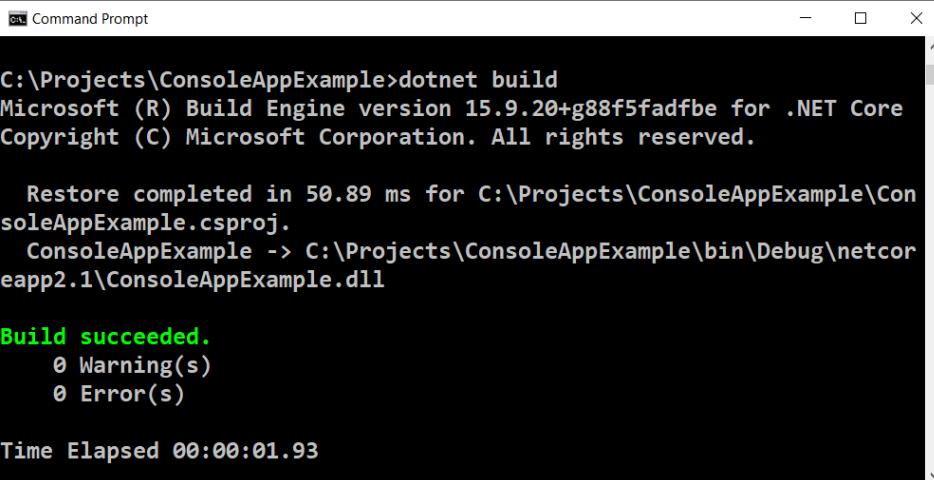
Programming Languages

A **programming language** is an artificial language (syntax for expression), meant for **giving commands** that we want the computer to read, process and execute. Using programming languages, we write sequences of commands (**programs**), which **define what the computer should do**. Examples of programming languages are C#, Java, JavaScript, Python, C, C++, PHP, Swift, Go and many others. These languages differ in their philosophy, syntax, purpose, programming constructions and execution environment. The execution of computer programs can be done with a **compiler** or with an **interpreter**.

Compilers

The **compiler** translates the code from programming language to **machine code**, as for each of the constructions (commands) in the code it chooses a proper, previously prepared fragment of machine code and in the meantime, it **checks the text of the program for errors**. Together, the compiled fragments comprise the program into a machine code, as the microprocessor of the computer expects

it. After the program has been compiled, it can be executed directly from the microprocessor in cooperation with the operating system. With compiler-based programming languages **the compilation of the program** is done obligatory before its execution, and syntax errors (wrong commands) are found during compile time. Languages like C++, C#, Java, Swift and Go work with a **compiler**. This is an example how the compiler execution may look (the console-based **dotnet** compiler):



```
Command Prompt

C:\Projects\ConsoleAppExample>dotnet build
Microsoft (R) Build Engine version 15.9.20+g88f5fadfbe for .NET Core
Copyright (C) Microsoft Corporation. All rights reserved.

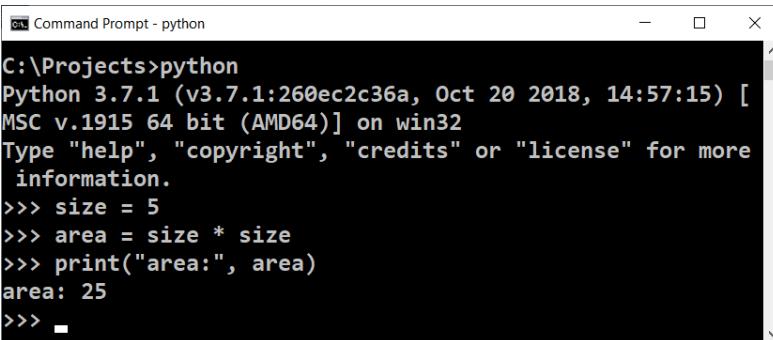
  Restore completed in 50.89 ms for C:\Projects\ConsoleAppExample\ConsoleAppExample.csproj.
  ConsoleAppExample -> C:\Projects\ConsoleAppExample\bin\Debug\netcoreapp2.1\ConsoleAppExample.dll

Build succeeded.
  0 Warning(s)
  0 Error(s)

Time Elapsed 00:00:01.93
```

Interpreters

Some programming languages do not use a compiler and are being **interpreted directly** by a specialized software called an "interpreter". The **interpreter** is "a program for executing programs", written in some programming language. It executes the commands in the program one after another, as it understands not only a single command and sequences of commands, but also other language constructions (evaluations, iterations, functions, etc.). Languages like Python, PHP and JavaScript work with an interpreter and are being executed without being compiled. Due to the absence of previous compilation, in interpreted languages **the errors are being found during the execution time**, after the program starts running, not previously. This is an example how an interpreter may look (the **python** interpreter in the console):



```
Command Prompt - python

C:\Projects>python
Python 3.7.1 (v3.7.1:260ec2c36a, Oct 20 2018, 14:57:15) [
MSC v.1915 64 bit (AMD64)] on win32
Type "help", "copyright", "credits" or "license" for more
information.
>>> size = 5
>>> area = size * size
>>> print("area:", area)
area: 25
>>> -
```

Development Environments (IDE)

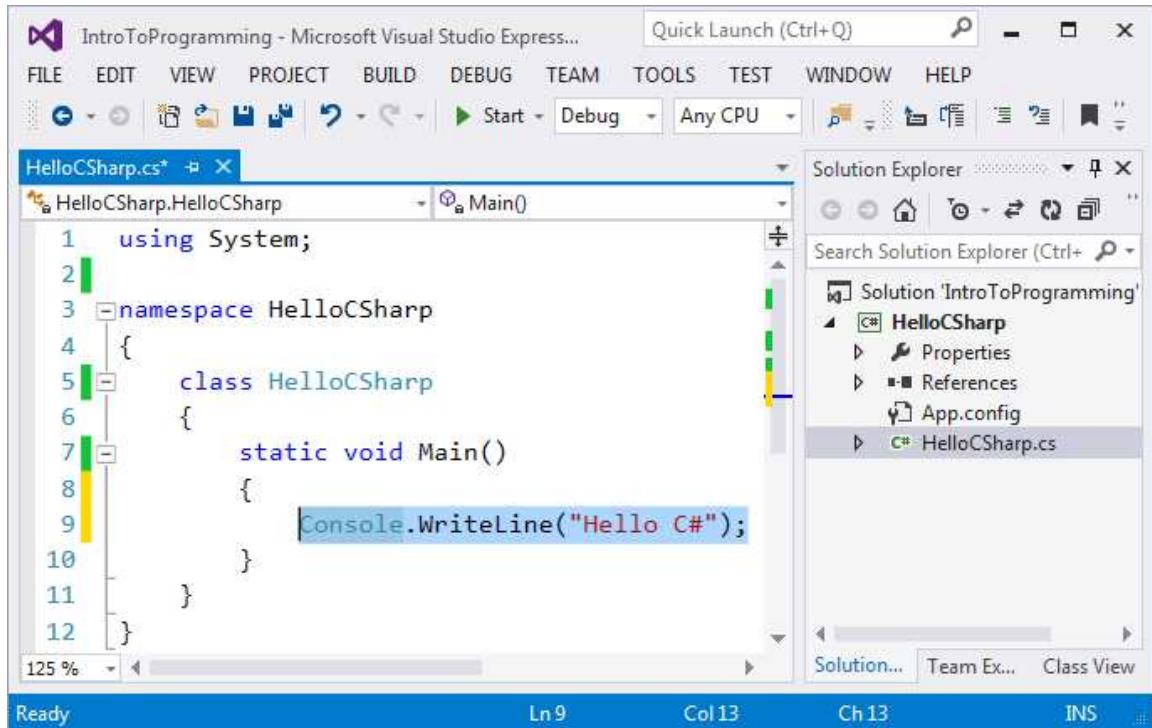
An **environment for development** (Integrated Development Environment – **IDE**) is a combination of traditional tools for development of software applications. In the development environment we write code, compile and execute the programs. Development environments integrate in them **a text editor** for writing code, **a programming language**, **a compiler or an interpreter** and **a runtime environment** for executing programs, **a debugger** for tracking the program and seeking out errors, **tools for user interface design** and other tools and add-ons.

Environments for development are convenient, because they integrate everything necessary for the development of the program, without the need to exit the environment. If we don't use an environment for development, we will have to write the code in a text editor, to compile it with a command on the console, to run it with another command on the console and to write more additional commands when needed, which is very time consuming. That is why most of the programmers use an IDE in their everyday work.

For programming with the **C# language** the most commonly used IDE is **Visual Studio**, which is developed and distributed freely by Microsoft and can be downloaded from: <https://www.visualstudio.com/downloads>. Alternatives of Visual Studio are:

- Rider – <https://www.jetbrains.com/rider>
- MonoDevelop / Xamarin Studio – <https://www.monodevelop.com>
- SharpDevelop – <http://www.icsharpcode.net/OpenSource/SD>
- Visual Studio Code - <https://code.visualstudio.com>
- Eclipse aCute – <https://projects.eclipse.org/projects/tools.acute>

In the current book, we are going to use the development environment **Visual Studio**. This is an example how a development IDE may look (the Visual Studio IDE for C#):



Runtime Environments, Low-Level and High-Level Languages

A program, in its essence, is a **sequence of instructions** that make the computer do a certain task. They are being entered by the programmer and **are being executed unconditionally by the machine**.

Video: Runtime Environments and Programming Languages

Watch a video lesson to learn about runtime environments and programming languages (high level and low level): <https://youtu.be/ziG5v36ISVk>.

Runtime Environments

Runtime environments are needed by some languages to execute the compiled programs. For example, the compiled **C#** programs are executed by the **.NET Core** runtime environment and the compiled **Java** programs are executed by **Java JRE** runtime environment. Other languages do not need compilation, but still require a runtime environment. For example, the **Python** programs are executed by the **Python interpreter** and runtime environment and the **JavaScript** programs are executed by the **Node.js** runtime environment or by a **Web browser** (which provides another JS runtime environment).

Programming Languages: Low-Level and High-Level

There are different kinds of **programming languages**. Via languages of the lowest level you can write **the instructions** that **manage the processor**, for example, using the "**assembler**" language. With a bit higher level languages, like **C** and **C++**, you can create an operating system, drivers for hardware management (for example a video card driver), web browsers, compilers, engines for graphics and games (game engines) and other system components and programs. With languages of even higher level, like **C#**, **Python** and **JavaScript** you can create application programs, for example a program for reading emails or a chat program.

Low level languages manage the hardware directly and require a lot of effort and a large count of commands to do a single task. **Languages of higher level** require less code for a single task, but do not have a direct access to hardware. Application software is developed using such languages, for example web applications and mobile applications.

Most of the software that we use daily, like music players, video players, GPS trackers, etc., are written with **languages for application programming** that are high-level, like **C#**, **Java**, **Python**, **C++**, **JavaScript**, **PHP** and others.

C# is a compiled language, which means that we write commands that are being compiled before they're being executed. Exactly these commands, through a help program (a compiler), are being transformed into a file, which can be executed (executable). To write a language like **C#** we need a text editor or a development environment and **.NET Runtime Environment** (like **.NET Core**).

.NET Runtime Environment

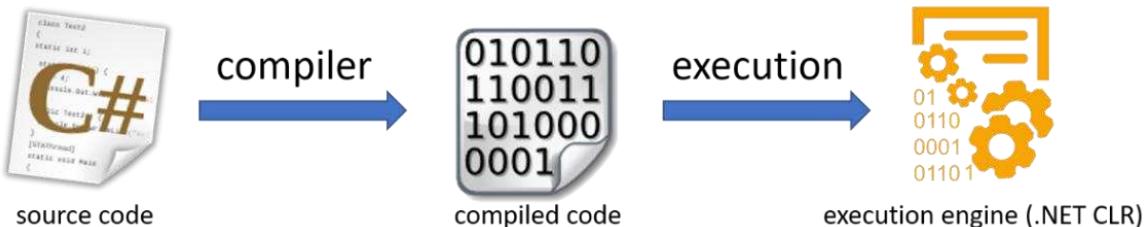
.NET Runtime Environment represents a virtual machine, something like a computer in the computer, which can run a compiled **C#** code. With the risk of going too deep into details, we have to explain that the language **C#** is compiled into an intermediary **.NET** code and is executed from the **.NET** environment, which compiles this intermediary code additionally into machine instructions (machine code) in order to be executed by the microprocessor. **.NET** environment contains libraries with classes, **CSC** compiler, **CLR** (Common Language Runtime – **CLR**) and other components, which are required for working with the language **C#** and run **C#** programs.

The **.NET environment** is available as a free software with open source code for every modern operating system (like Windows, Linux and Mac OS X). It has two variations, **.NET Framework** (the older one) and **.NET Core** (the newer one), but none of that is essential when it comes to getting into programming. Let us focus on writing programs with the **C#** language.

Compilation and Execution of C# Programs

As we have already mentioned, a program is **a sequence of commands**, otherwise said, it describes a sequence of calculations, evaluations, iterations and all kinds of similar operations, which aim to obtain certain result.

A **C#** program is written in a text format, and the text of the program is called a **source code**. It gets compiled into an **executable file** (for example **Program.cs** gets compiled to **Program.exe**) or it is **executed directly** from the **.NET** environment.



The process of **compilation** of the code before its execution is used only in compiled languages like C#, Java and C++. With **scripts and interpreted languages**, like JavaScript, Python and PHP, the source code gets executed **step by step** by an interpreter.

Computer Programs – Examples

Let's start with a few simple examples of short C# programs.

Video: Computer Programs – Examples

Watch a video lesson about the explained below sample computer programs: <https://youtu.be/TIwcDNJFid4>.

Example: A Program That Plays the Musical Note "A"

Our **first program** is going to be a single **C# command** that plays the musical note "A" (432 Hz) with a duration of half a second (500 milliseconds):

```
Console.Beep(432, 500);
```

A bit later we will find out how we can execute this command and hear the sound of the note, but for now let's just look at what the commands in programming represent. Let's get to know a couple more examples.

Example: A Program That Plays Musical Notes

We can complicate the previous program by giving for execution **repeating commands in a loop** for playing a sequence of notes with rising height:

```
for (i = 200; i <= 4000; i += 200)
{
    Console.Beep(i, 100);
}
```

In the example above we made the computer play one after another for a very short time (100 milliseconds) all the notes with height 200, 400, 600 etc. Hz until they reach 4000 Hz. The result of the program is playing something like a **melody**.

How do iterations (cycles) work in programming? We will learn that in the chapter "[Loops](#)", but for now just accept that we repeat some command many times.

Example: A Program That Converts USD to EUR

Let's look at another simple program that reads from the user some **amount** of money in U.S. Dollars (USD), an integer, converts it into **Euro** (EUR) by **dividing it** by the Euro's rate and **prints** the obtained result. This is a program of 3 consecutive commands:

```
var dollars = int.Parse(Console.ReadLine());
var euro = dollars * 0.883795087;
Console.WriteLine(euro);
```

Run the above code example: <https://repl.it/@nakov/dollars-to-euro-converter-csharp>.

We examined **three examples of computer programs**: a single command, series of commands in a loop and 3 consecutive commands. Now let's get to the more interesting part: how we can write our own programs in **C#** and how we can compile them and run them.

How to Write a Console Application?

As a next step, let's pass through the steps of creating and executing a computer program that reads and writes its data from and on the **text console** (a window for entering and printing text). These programs are called "**console programs**". But before that, we should **install and prepare the development environment**, in which we are going to write and run the C# programs from this book and the exercises in it.

Development Environments (IDE) and Visual Studio

As it has already been said, in order to program we need an **Integrated Development Environment (IDE)**. This is actually an editor for programs, in which we write the program code and we can compile it and run it to see the errors, fix them and start the program again.

- For programming with C# we use **Visual Studio** IDE for Windows operating system and **MonoDevelop** or **Rider** for Linux or Mac OS X.
- If we program with Java, the environments **IntelliJ IDEA**, **Eclipse** or **NetBeans** are suitable.
- If we write in Python, we can use the **PyCharm** environment.

Video: Installing and Running Visual Studio

Watch the following video lesson for guidelines about how to install and run the Visual Studio IDE: <https://youtu.be/6AhALTJEagA>.

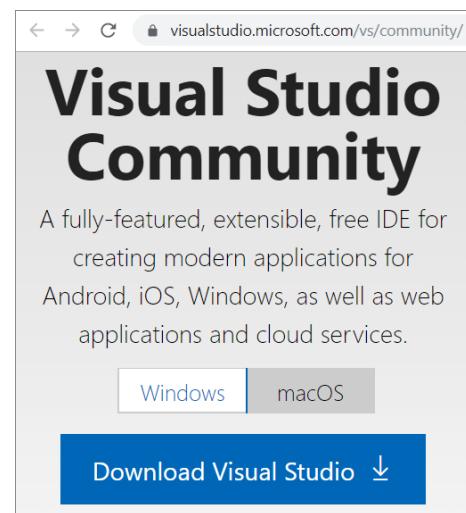
Installing Visual Studio

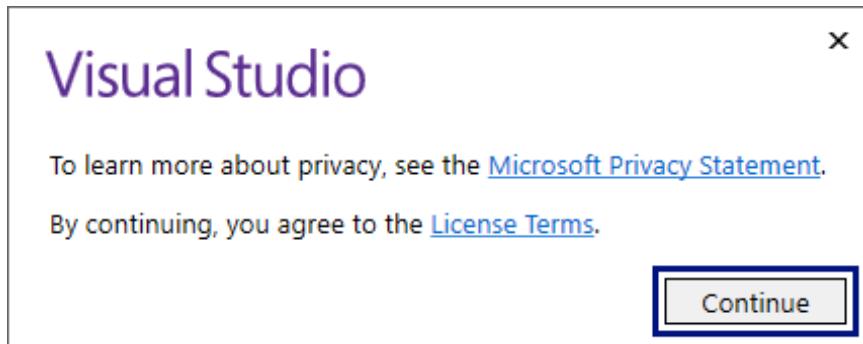
We begin with the installation of the integrated environment **Microsoft Visual Studio** (Community edition, version 2017). Installing later versions of Visual Studio (like Visual Studio 2019 and Visual Studio 2021) should be very similar.

The **Community** version of Visual Studio (VS) is distributed freely by Microsoft and can be downloaded from: <https://www.visualstudio.com/vs/community/>.

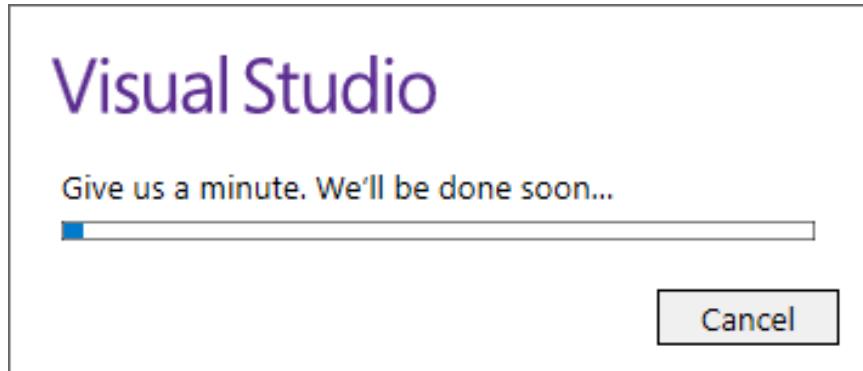
The installation is typical for Windows with **[Next]**, **[Next]** and **[Finish]**, but it's important to include the components for "**desktop development**" and "**ASP.NET**". It is not necessary to change the rest of the settings for the installation.

The next lines describe in detail **the steps for the installation of Visual Studio** (version Community 2017). After we download the installation file and start it, the following screen appears:

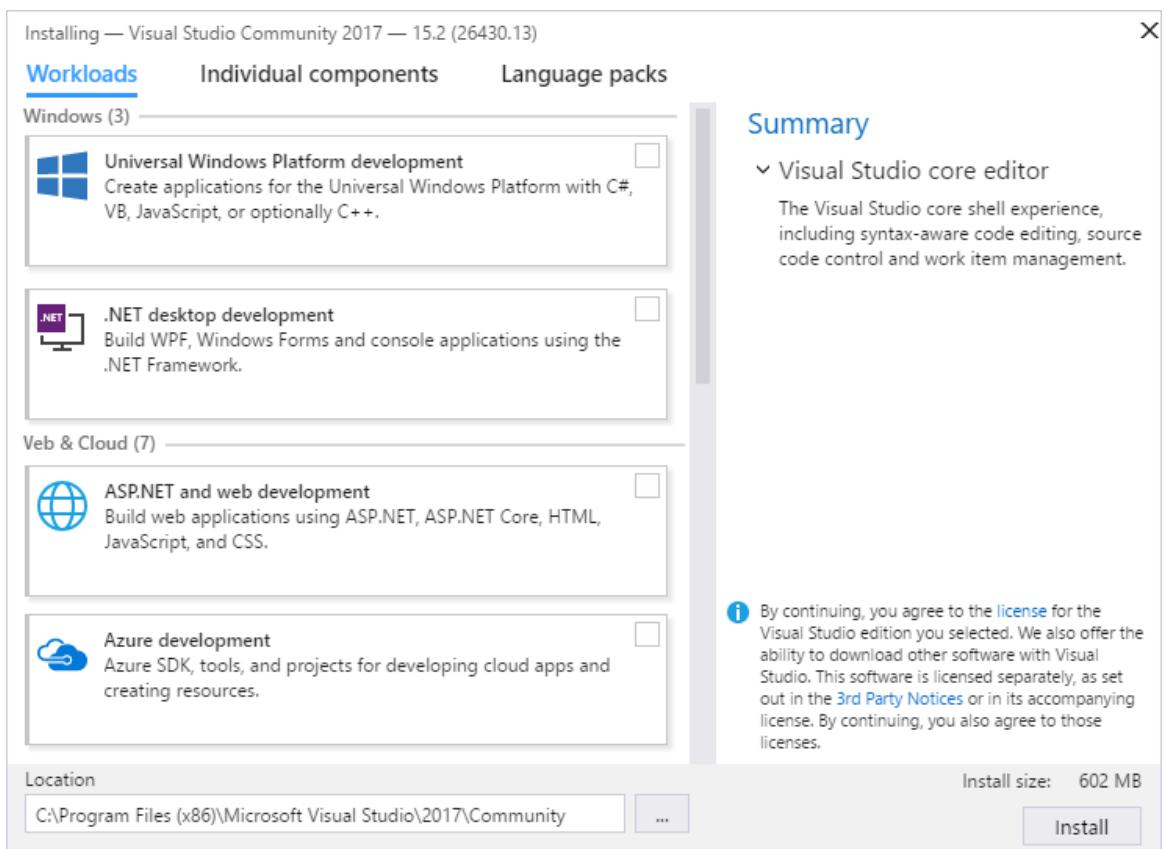




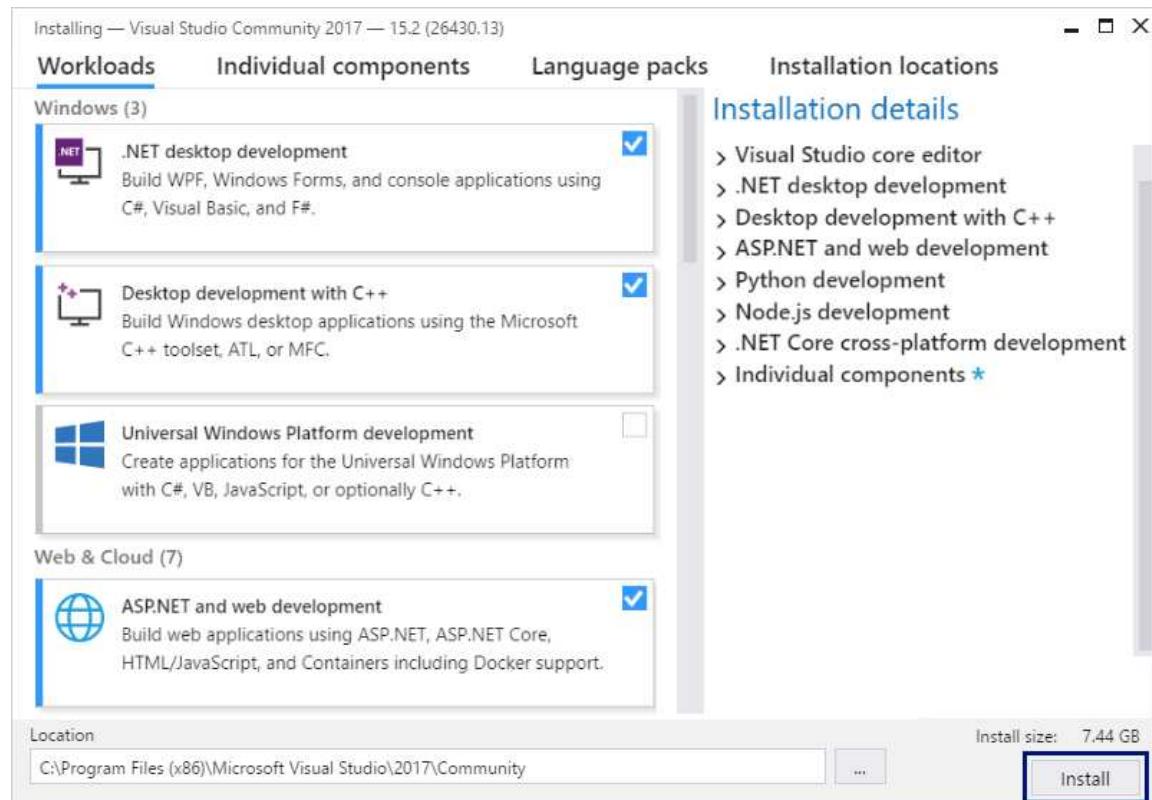
Press the [Continue] button and you will see the screen below:



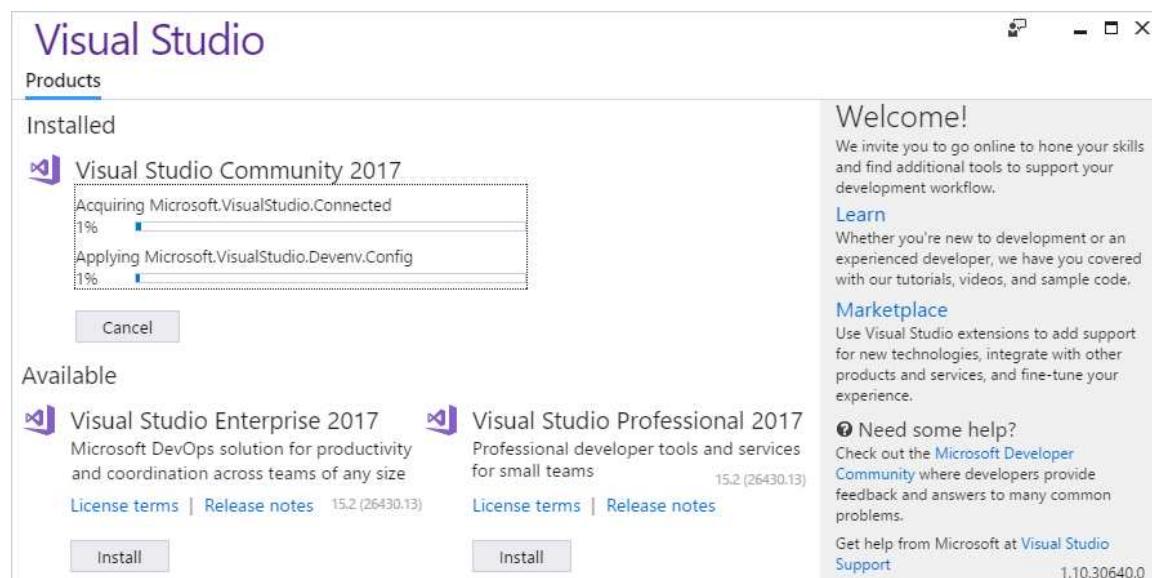
A screen with the installation panel of Visual Studio is being loaded.



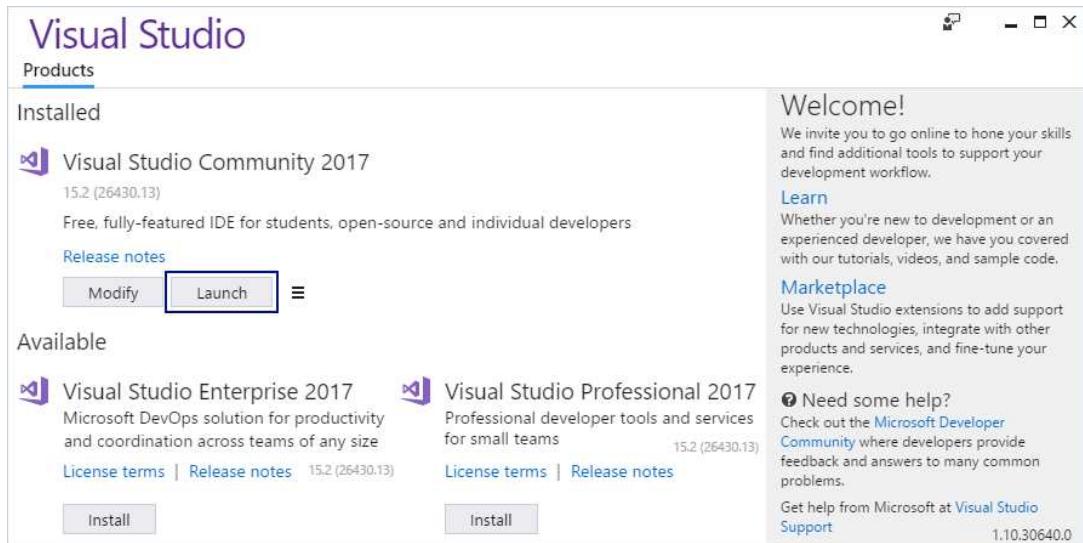
Put a check mark on [Universal Windows Platform development], [.NET desktop development] and [ASP.NET and web development], then press the [Install] button. Basically, this is everything.



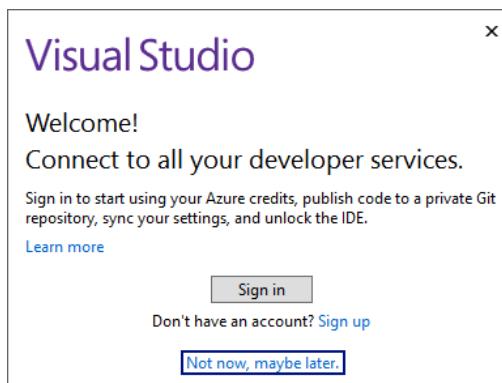
The installation of Visual Studio begins, and a screen like the one below will appear:



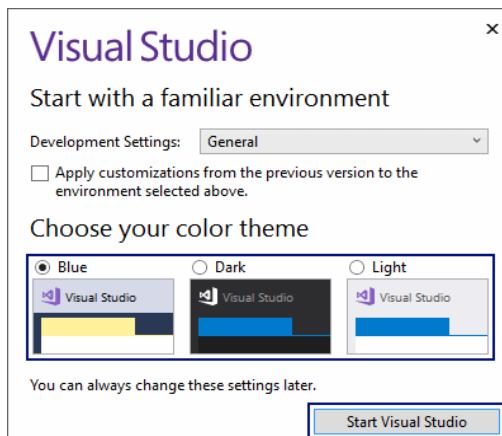
After Visual Studio is installed, an informative screen will appear. Press the [Launch] button to start it.



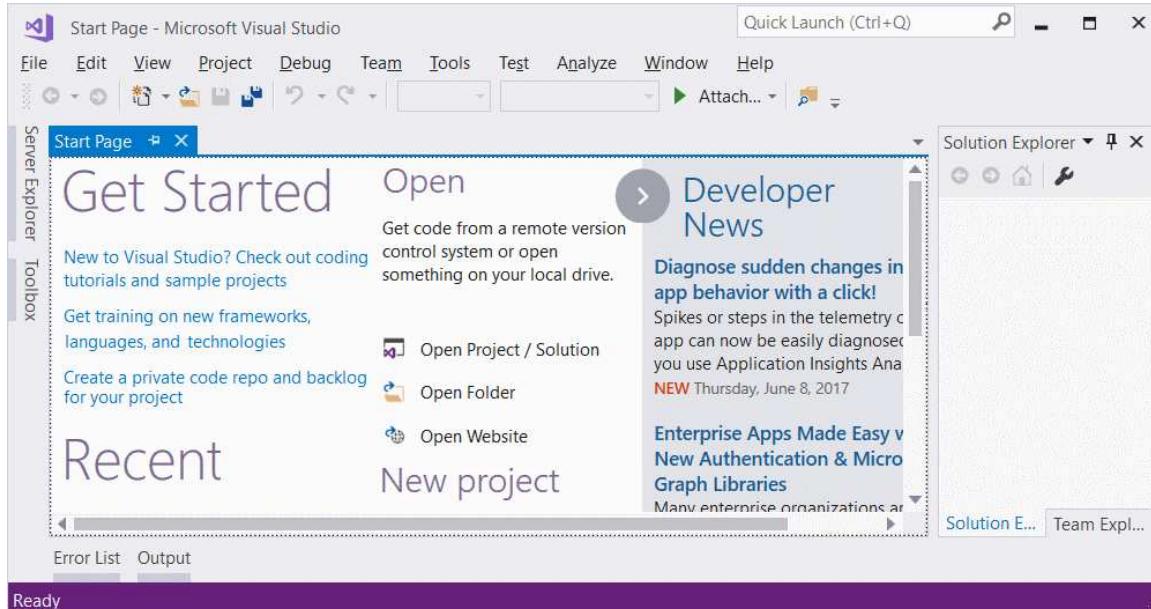
Upon **starting VS** a screen appears like the one below. On it you can choose whether you will enter Visual Studio using a Microsoft account. For now, we choose to continue without being logged into our Microsoft account, and therefore we choose the option [**Not now, maybe later.**]. At a later point, if you have such an account, you may log in, and if you don't have one, and you have difficulties with its creation, you can always ask in the SoftUni official [discussion forum](http://forum.softuni.org) (<http://forum.softuni.org>) or in the SoftUni official [Facebook page](https://fb.com/softuni.org) (<https://fb.com/softuni.org>).



The next step is to choose **the color theme**, in which Visual Studio is visualized. The choice here lays completely on the preferences of the user and it doesn't matter which option will be chosen.



Press the [Start Visual Studio] button and the main view of Visual Studio Community will be displayed:



That's all. We are ready to work with Visual Studio.

Older Versions of Visual Studio

You can use older versions of Visual Studio (for example version 2015 or 2013 or even 2010 or 2005), but it is not recommended, as they don't contain some of the newer options for development, and not all the examples from the book will run the same way.

Online Development Environments

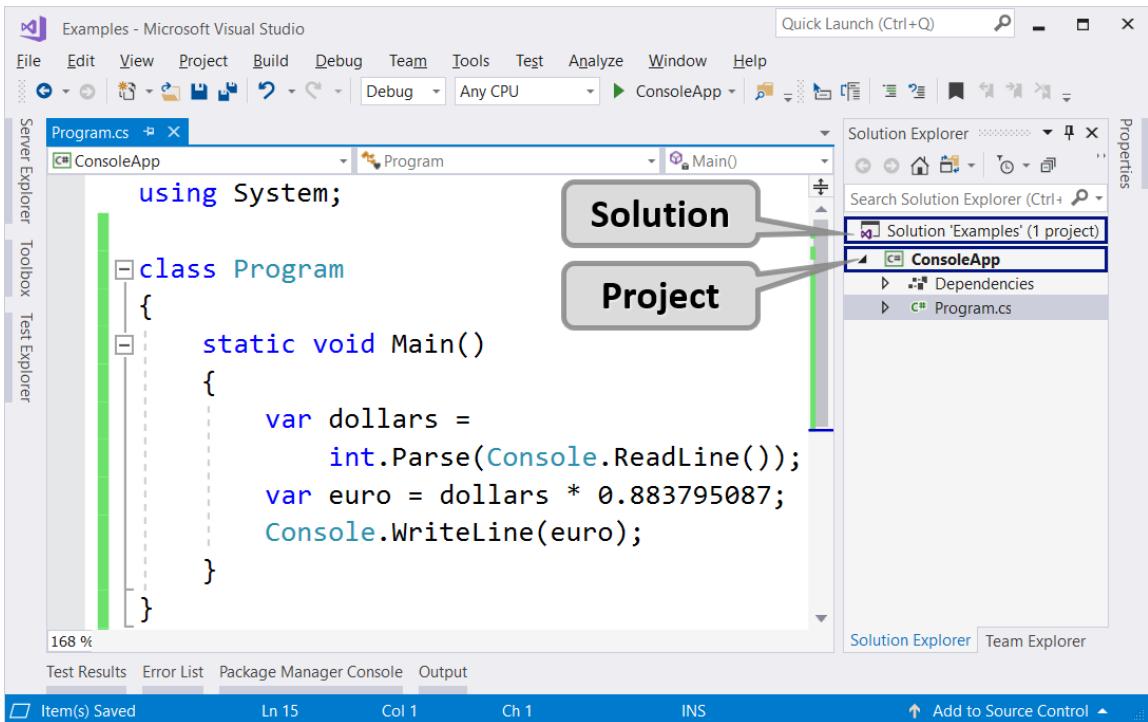
There are alternative environments for development online directly into your web browser. These environments are not very convenient, but if you don't have other opportunity, you can start your training with them and install Visual Studio later. Here are some useful links:

- For the language C# the site .NET Fiddle allows code writing and its execution online: <https://dotnetfiddle.net>.
- For Java you can use the following online Java IDE: <https://www.compilejava.net>.
- For JavaScript you can write a JS code directly in the console of a given browser when you press [F12].
- The site Repl.it provides online coding environment for multiple languages (C#, Java, JS, Python, C++ and many more): <https://repl.it>.

Project Solutions and Projects in Visual Studio

Before we start working with Visual Studio, it is necessary to get familiar with the concepts of a **Visual Studio Solution** and a **Visual Studio Project**, which are an inevitable part of it.

Visual Studio Project represents "the project" we are working on. In the beginning, these will be our console applications, which we are going to learn writing with the help of the current book, the resources in it and the course Programming Basics in SoftUni. With deeper learning, time and practice, these projects will move into the direction of desktop applications, web applications and other developments.



A project in VS **logically groups multiple files** constructing a given application or a component. A **C# project** contains one or more **C# source files**, configuration files and other resources. In every C# source file, there is one or more **definition of types** (classes or other definitions). In **the classes** there are **methods** (actions), and they contain **a sequence of commands**. It sounds complicated, but with bigger projects a structure like this is very convenient and allows good organization of the work files.

Visual Studio **Solution** represents a container (a work **solution**), in which **a few projects are logically bound**. The purpose of the binding of these VS Projects is to create an opportunity for the code from any of the projects to collaborate with the code from the rest of the VS projects, to ensure the application or the website to work correctly. When the software product or service that we develop is big, it is built as a **VS Solution**, and this Solution is split into **projects** (VS Projects) and inside each project there are **folders with source files**. This hierarchical organization is much more convenient with more serious projects (let's say over 50 000 lines of code).

For **smaller projects** VS Solutions and VS Projects are **complicating the work**, rather than helping, but you will get used to it quickly.

Example: Creating a Console Application "Hello C#"

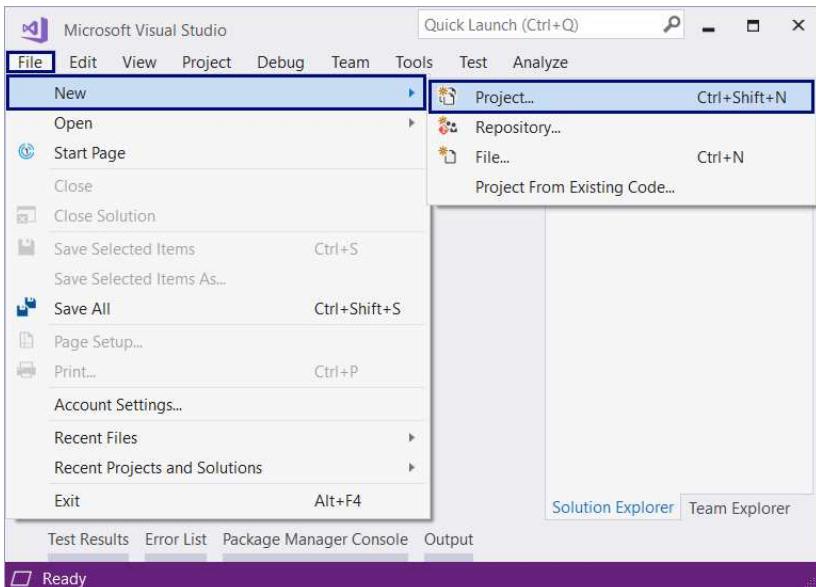
Let's create our first **console program** in Visual Studio. We will start the Visual Studio IDE, will create a new console-based C# project, will write a few lines of C# code and will compile and run the program. Finally, we will submit our C# code for evaluation in the automated Judge system.

Video: Console Application in Visual Studio

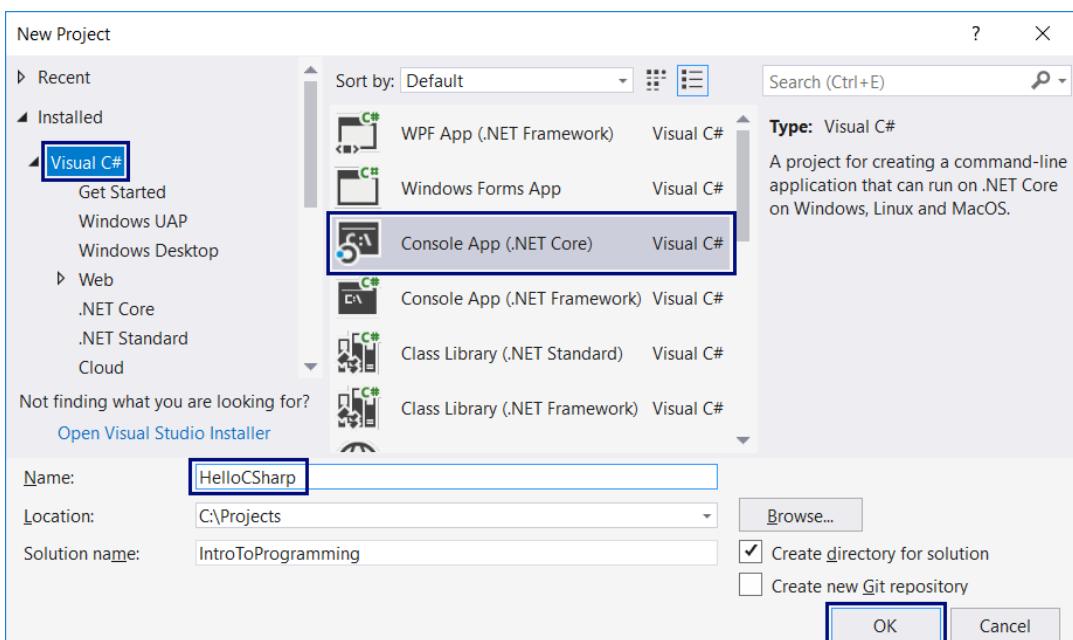
Watch a video lesson about creating a console app in Visual Studio: <https://youtu.be/ecAXCijk6Nw>.

Console App in Visual Studio: Step by Step

We already have Visual Studio and we can start it. Then, we create a new console project: [File] → [New] → [Project] → [Visual C#] → [Windows] → [Console Application].



We set a meaningful name to our program, for example `HelloCSharp`:



Visual Studio is going to create for us an empty C# program, which we have to finish writing (VS Solution with VS Project in it, with C# source file in it, with C# class in it, with `Main()` method in it).

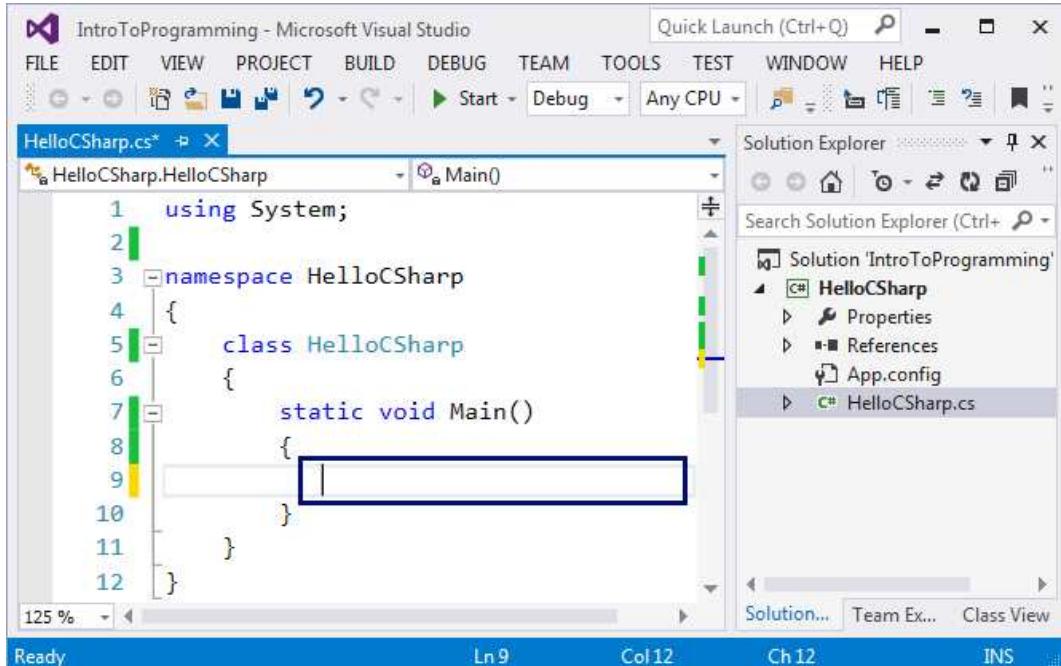
Writing the Program Code

The source code of the C# program is written in the section `Main(string[] args)`, between the opening and the closing parentheses `{ }`. This is the main method (action), that is being executed with the start of a C# program. This `Main()` method can be written in two ways:

- `static void Main(string[] args)` – with parameters from the command line (we are not going into details).
- `static void Main()` – without parameters from the command line.

Both ways are valid, as **the second one is recommended**, because it is shorter and clearer. By default, though, when creating a console application, Visual Studio uses the first way, which we can edit manually if we want to, and delete the part with the parameters `string[] args`.

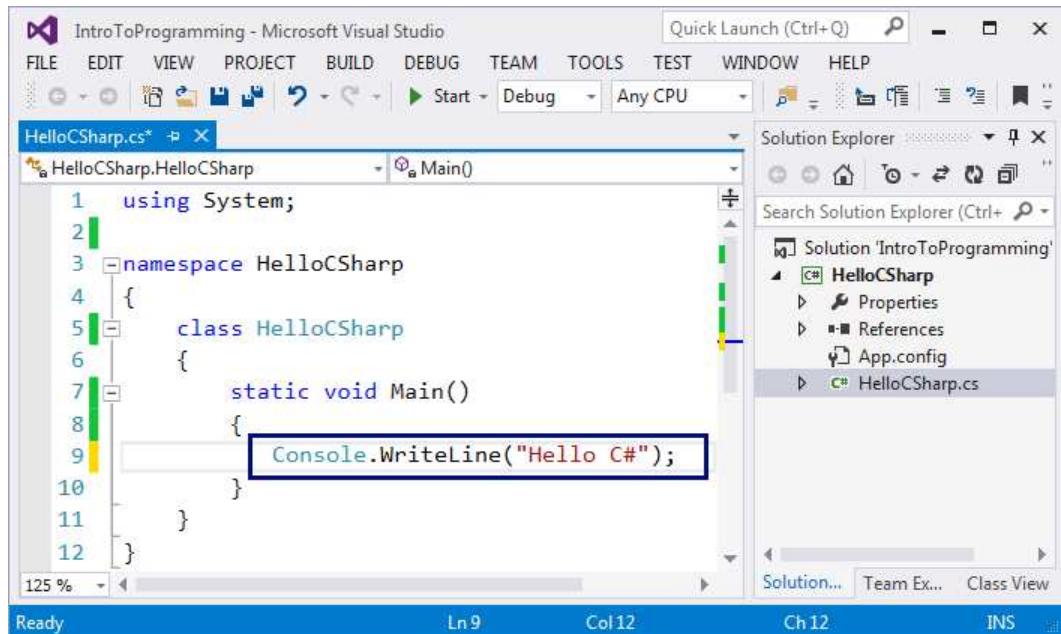
Press [Enter] after the opening parentheses `{` and **start writing**. The code of the program is written **inwards**, as this is a part of shaping up the text for convenience during a review and/or debugging.



Write the following command:

```
Console.WriteLine("Hello C#");
```

Here is how our program should look like in Visual Studio:



The command `Console.WriteLine("Hello C#")` in the C# language means to execute printing (`WriteLine(...)`) on the console (`Console`) and to print the text message `Hello C#`, which we should surround by quotation marks, in order to clarify that this is a text. In the end of each command in the C# language the symbol `;` is being put and it says that the command ends in that place (it doesn't continue on the next line).

This command is very typical in programming: we say a given **object** should be found (in this case the console) and some **action** should be executed upon it (in this case it is printing something that is given inside the brackets). More technically explained, we call the method `WriteLine(...)` from the class `Console` and give as a parameter to it a text literal "`Hello C#`".

Starting the Program

To start the program, press **[Ctrl + F5]**. If there aren't any errors, the program will be executed. The result will be written on the console (in the black window):

```
C:\WINDOWS\system32\cmd.exe
Hello C#
Press any key to continue . . .
```

Notice that we start it with **[Ctrl+F5]**, and not only **[F5]** or with the start button in Visual Studio. If we use **[F5]**, the program will run shortly and right afterwards the black window will disappear, and we are not going to see the result.

Actually, the output from the program is the following text message:

```
Hello C#
```

The message "`Press any key to continue . . .`" is displayed additionally on the last line on the console after the program ends, in order to push us to see the result from the execution and to press a key to close the console.

Testing the Program in the Judge System

Testing of the problems in this book is automated and is done through the Internet, using the **SoftUni Judge System**: <https://judge.softuni.org> website.

The **evaluation** of the submitted solutions is done immediately by the system. Each task goes through a sequence of tests, as every successfully passed test gives the points assigned for it. The tests that are applied to the tasks are hidden.

We can test the above program here: <https://judge.softuni.org/Contests/Practice/Index/503#0>. We place the source code of the program in the black field and we choose C# code, like it is shown on the figure.

We send our solution for evaluation using the **[Send]** button. The system gives a

Points	Time and memory used	Submission date
0 / 100	Memory: 7.09 MB Time: 0.031 s	18:01:10 28.01.2019

result back in a few seconds in the table with sent solutions. When necessary, we can press the button for renewing the results [refresh] in the upper right side of the table with sent solutions:

Submissions		
1	Refresh 	
Points	Time and memory used	Submission date
✓ 100 / 100	Memory: 7.06 MB Time: 0.015 s	18:01:32 28.01.2019 
✗ 0 / 100	Memory: 7.09 MB Time: 0.031 s	18:01:10 28.01.2019 

In the table with the sent solutions the judge system is going to show one of the following **possible results**:

- **Points count** (between 0 and 100), when the submitted code is compiled successfully (there are no syntax errors) and can be tested.
 - When the **solution is correct** all of the tests are marked in green and we get **100 points**.
 - When the **solution is incorrect** some of the tests are marked in red and we get incomplete or 0 points.
- When the program is incorrect, we will get **an error message** upon compiling.

How to Register in SoftUni Judge?

Use your credentials (username + password) for the site **softuni.org / softuni.bg**. If you don't have a SoftUni registration, create one. It takes only a minute – a standard registration in an Internet site.

Testing the Programs That Play Notes

Now, after **you know how to run programs**, you can test your example programs that play musical notes. Have some fun, try these programs out. Try to change them and play with them. Change the command `Console.WriteLine("Hello C#");` with the command `Console.Beep(432, 500);` and start the program. Check if the sound of your computer is on and whether it's turned up. If you work in an online environment, you will not hear a sound, because the program is not executed on your computer, but elsewhere. This program cannot be checked in the SoftUni judge.

Typical Mistakes in C# Programs

Now we will review the **typical mistakes in the C# programs** of the beginners, like missing semicolon, missing quotations mark, missing parenthesis, wrong letter capitalization, etc.

Video: Typical Mistakes in C# Programs

Watch a video lesson about the most typical mistakes in the C# programs of the beginners: <https://youtu.be/8XwM2AVC0wU>.

Writing Outside if the Main Method

One of the common mistakes with beginners is **writing outside the body of the `Main()` method**, because the integrated environment or the compiler can't read the given commands in the program

correctly. Here is an example for an incorrectly written program, where the command are placed outside of the **Main()** method:

```
static void Main(string[] args)
{
}
Console.WriteLine("Hello C#");
```

Wrong Letter Capitalization

Another mistake is switching **capital and small letters**, and these matter for calling the commands and their correct functioning. Here is an example of such a mistake:

```
static void Main(string[] args)
{
    Console.Writeline("Hello C#");
}
```

In the example above **Writeline** is written wrong and has to be fixed to **WriteLine**.

Missing Semicolon

The absence of **a semicolon** (**;**) in the end of the commands is one of the eternal problems of the beginner programmer. Skipping this sign leads to **incorrect functioning of the program** and often the **problem stays unnoticed**. Here is an example of a mistaken code:

```
static void Main(string[] args)
{
    Console.Writeline("Hello C#")
}
```

Missing or Wrong Quotation Mark or Parenthesis

Missing **quotation mark** or **the absence of opening or closing parentheses** can also turn out to be a problem. Same as the semicolon, here also the problem leads to **incorrect functioning of the program** or overall to its failure. This mistake is hardly noticeable in a larger code. Here is an example of a program with errors:

```
static void Main(string[] args)
{
    Console.WriteLine("Hello C#");
}
```

This program will throw **a compile time error** and the build is going to fail, and even before that the code will become underlined, in order to point the programmer to the mistake that they'd made (the missing closing quotation mark):

The screenshot shows the Visual Studio IDE. In the top window, titled 'Program.cs*', the code is:

```

1  using System;
2
3  class HelloCSharp
4  {
5      static void Main()
6      {
7          Console.WriteLine("Hello C#");
8      }
9

```

The 'Error List' window below shows three errors:

Code	Description	Project	File	Line
✖ CS1026) expected	HelloCSharp	Program.cs	7
✖ CS1002	; expected	HelloCSharp	Program.cs	7
✖ CS1010	Newline in constant	HelloCSharp	Program.cs	7

Another example is missing { or }. It may produce unexpected error messages, not always easy to understand.

```

class Example
{
    static void Main(string[] args)
    {
        Console.WriteLine("Hello C#");
    }
}

```

Exercises: First Steps in Coding

Welcome to the **exercises**. Now we are going to write a couple of console applications, by which we are going to make a few more steps into programming. After that we will show how we can program something more complex – programs with graphical user interface.

Video: Chapter Summary

Watch the following short video to summarize what we learned about coding in this chapter:
<https://youtu.be/GstN43-eN2g>.

What We Learned in This Chapter?

First, we have learned **what is programming** – giving commands, written in a computer language, which the machine understands and is able to execute. We understood what a **computer program** is – it represents a **sequence of commands**, arranged one after another. We got familiar with the **language for programming C#** on a base level and how **to create simple console applications** with Visual Studio. We followed the **structure of the programming code** in the C# language, as for example, the fact that commands are mainly given in the section **static void Main(string[] args)** between the **opening and closing curly parentheses**. We saw how to print with **Console.WriteLine(...)** and how to start our program with [Ctrl + F5]. We learned how to test our code in **SoftUni Judge**.

The Exercises

Let's get started with the **exercises**. You didn't forget that programming is learned by writing a lot of code and solving problems, did you? Let's solve a few problems to confirm what we have learned.

Problem: Expression

Write a console-based C# console program that **calculates** and **prints** the value of the following numerical expression:

$$(3522 + 52353) * 23 - (2336 * 501 + 23432 - 6743) * 3$$

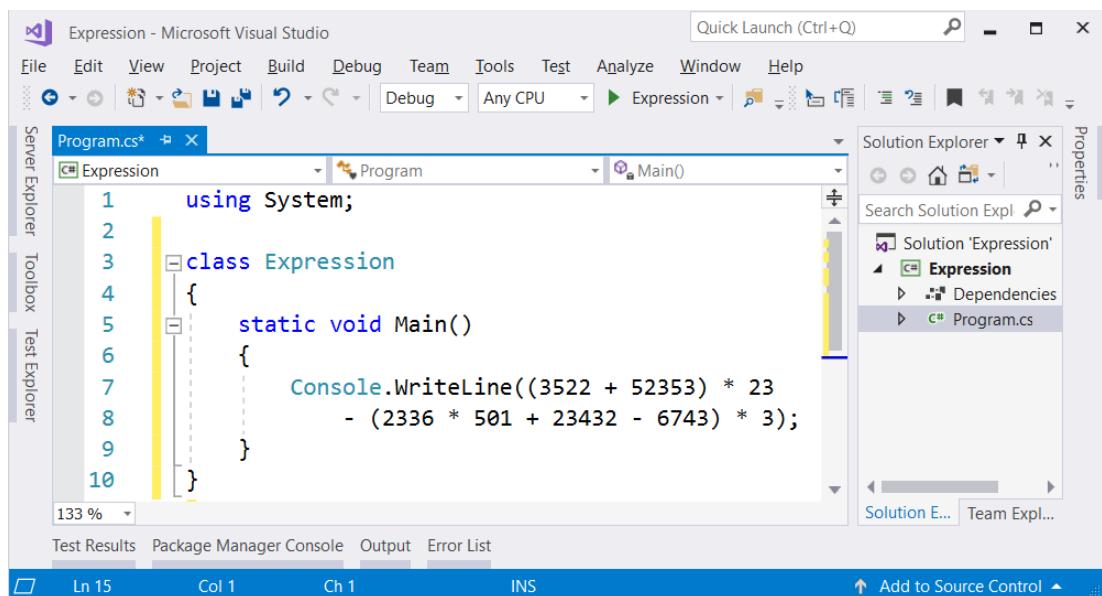
Note: it is not allowed to previously calculate the value (for example with Windows Calculator).

Video: Problem "Expression"

Watch a video lesson to learn how to solve the "Expression" problem step by step, with concise explanations: <https://youtu.be/JEbwS2xtbLw>.

Hints and Guidelines

Create a new C# console project with title "Expression". Find the method `static void Main(string[] args)` and go into its **body** between { and }. After that, **write the code** that calculates the above numerical expression and prints its value on the console. Put the above numerical expression inside the brackets of the command `Console.WriteLine(...)`:



Start the program with [Ctrl+F5] and check if the result is the same as the one in the picture:



Testing in the Judge System

Test your solution here: <https://judge.softuni.org/Contests/Practice/Index/503#1>.

```

1 using System;
2
3 class Expression
4 {
5     static void Main()
6     {
7         Console.WriteLine((3522 + 52353) * 23 - (2336*501 + 23432 - 6743) * 3);
8     }
9 }
```

Allowed working time: 0.100 sec.
Allowed memory: 16.00 MB
Size limit: 16.00 KB
Checker: Numbers Checker

Submissions		
Points	Time and memory used	Submission date
✓ 100 / 100	Memory: 7.22 MB Time: 0.015 s	14:04:00 06.06.2017

Problem: Numbers from 1 to 20

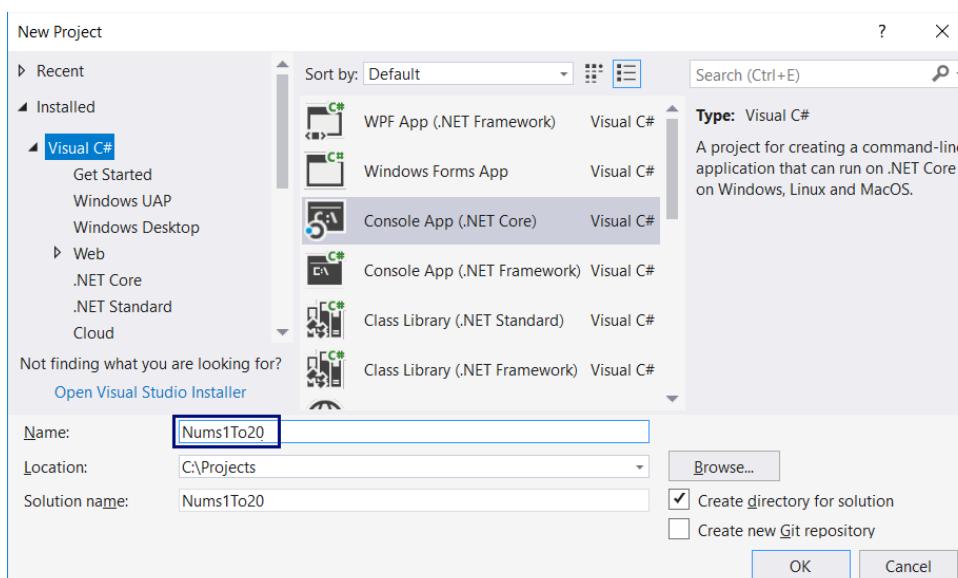
Write a C# console program that prints the numbers from 1 to 20 on separate lines on the console.

Video: Problem "Numbers from 1 to 20"

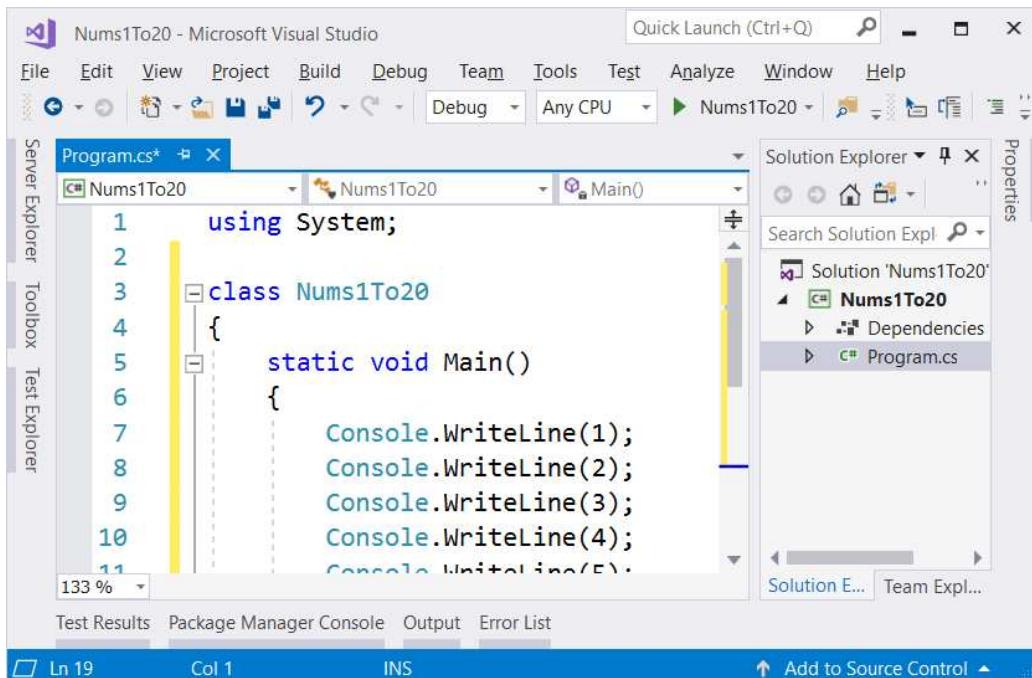
Watch a video lesson to learn how to solve the "Numbers from 1 to 20" problem step by step:
<https://youtu.be/8Qne7CBM2SQ>.

Hints and Guidelines

Create a C# console application with name "Nums1To20":



Inside the `static void Main()` method write 20 commands `Console.WriteLine()`, each on a separate line, in order to print the numbers from 1 to 20 one after another. Some of you may be wondering if there is a **smarter way**. Relax, there is, but we will mention it later on.



Now **we start the program** and we check if the result is what it is supposed to be:

```
1
2
...
20
```

Testing in the Judge System

Test your solution here: <https://judge.softuni.org/Contests/Practice/Index/503#2>.

Now think whether we can write the program **a smarter way**, so we don't repeat the same command 20 times. Seek out information on the Internet about "[for loop C#](#)".

Problem: Triangle of 55 Stars

Write a C# console program that prints a triangle made of 55 stars on 10 lines:

```
*
```

~~**~~
~~***~~
~~****~~
~~*****~~
~~*****~~
~~*****~~
~~*****~~
~~*****~~

Video: Problem "Triangle of 55 Stars"

Watch a video lesson to learn how to solve the "Triangle of 55 Stars" problem step by step:
<https://youtu.be/BfITRoOQYLA>.

Hints and Guidelines

Create a new console C# application with name "**TriangleOf55Stars**". Inside it, write code that prints the triangle of stars, for example through 10 commands, as the ones pointed out below:

```
Console.WriteLine("*");
Console.WriteLine(" **");
...
...
```

Testing in the Judge System

Test your solution here: <https://judge.softuni.org/Contests/Practice/Index/503#3>.

Try to **improve your solution**, so that it doesn't have many repeating commands. Could it be done with a **for** loop? Did you find a smart solution (for example with a loop) of the previous task? With this task you can also use something similar, but a bit more complex (two loops, one inside the other). If you don't succeed, there is no problem, we will be learning loops in a few chapters and you will be reminded of this task then.

Problem: Calculate Rectangle Area

Write a C# program that **reads** from the console two numbers, **a** and **b**, **calculates** and **prints** the area of a rectangle with sides **a** and **b**.

Sample Input and Output

a	b	area
2	7	14

a	b	area
12	5	60

a	b	area
7	8	56

Video: Problem "Rectangle Area"

Watch a video lesson to learn how to solve the "Rectangle Area" problem in C# step by step:
<https://youtu.be/6fwNJ5k9zTE>.

Hints and Guidelines

Create a new console C# program. To read both of numbers, use the following commands:

```
static void Main(string[] args)
{
    var a = decimal.Parse(Console.ReadLine());
    var b = decimal.Parse(Console.ReadLine());
    // TODO: calculate the area and print it at the console
}
```

What remains is to finish the program above, to calculate the area of the rectangle and to print it. Use the command that is already known to us **Console.WriteLine()** and put inside its brackets the multiplication of the numbers **a** and **b**. In programming, multiplication is done using the operator *****.

Test Your Solution

Test your solution with a few examples. You have to get an output, similar to this one (we enter 2 and 7 as input and the program prints result 14 – their multiplication):

```
2
7
14
```

Testing in the Judge System

Test your solution here: <https://judge.softuni.org/Contests/Practice/Index/503#4>.

* Problem: A Square Made of Stars

Write a C# console program that **reads** from the console **an integer N** and **prints** on the console **a square made out of N stars**, like in the examples below.

Sample Input and Output

Input	Output	Input	Output	Input	Output
3	*** * * ***	4	**** * * * * ****	5	***** * * * * * * *****

Video: Problem "Square of Stars"

Watch a video lesson to learn how to solve the "Square of Stars" problem step by step:
<https://youtu.be/zai-DRbaHal>.

Hints and Guidelines

Create a new **console C# program**. To read the number N ($2 \leq N \leq 100$), use the following code:

```
static void Main(string[] args)
{
    var n = int.Parse(Console.ReadLine());
    // TODO: Print the rectangle
}
```

Finish the program above, so that it prints a square, made out of stars. It might be necessary to use **for** loops. Look for information on the Internet.

Attention: this task is harder than the rest and is given now on purpose, and it's marked with a star, in order to provoke you to look for information on the Internet. This is one of the most important skills that you have to develop while you're learning programming: **looking for information on the Internet**. This is what you're going to do every day, if you work as a developer, so don't be scared, try it out.

If you have any difficulties, you can also ask for help in the SoftUni official **discussion forum** (<http://forum.softuni.org>) or in the SoftUni official **Facebook page** (<https://fb.com/softuni.org>).

Testing in the Judge System

Test your solution here: <https://judge.softuni.org/Contests/Practice/Index/503#5>.

Lab: Graphical and Web Applications

Now we are about to build one simple **web application** and one simple **graphical application**, in order to take a look at what we will be able to create when we progress with programming and software development. We are not going to look through the details about the used techniques and constructions fundamentally. Rather than that, we are just going to take a look at the arrangement and functionality of our creation. After we progress with our knowledge, we will be able to do bigger and more complex software applications and systems. We hope that the examples given below will **straighten your interest**, rather than make you give up.

Console, Graphical and Web Applications

With **console applications**, as you can figure out yourselves, **all operations** for reading input and printing output are done through the **console**. The **input data is entered** in the console, which is then read by the application, also in it, and the **output data is printed** on the console after or during the runtime of the program.

While a console application **uses the text console**, web applications **use web-based user interface**. To **execute them**, two things are needed – a **web server** and a **web browser**, as the **browser** plays the main role in the **visualization of the data and the interaction with the user**. Web applications are much more pleasant for the user, they visually look better, and a mouse and touch screen can be used (for tablets and smartphones), but programming stands behind all of that. And this is why **we have to learn to program** and we have already made our first very little steps towards that.

Graphical (GUI) applications have **a visual user interface**, directly into your computer or mobile device, without a web browser. Graphical applications (also known as desktop applications) **contain one or more graphical windows**, in which certain **controllers** are located (text fields, buttons, pictures, tables and others), **serving for dialog** with the user in a more intuitive way. Similar to them are the mobile applications in your telephone or your tablet: we use forms, text fields, buttons and other controls and we control them by programming code. This is why we are learning how to write code now: **the code is everywhere in software development**.

Exercises: GUI and Web Applications

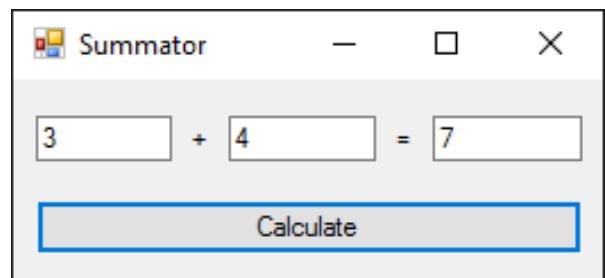
In the next exercises we will create a **GUI** and a **Web** application:

- Graphical Application "Summator" (Calculator)
- Web Application "Summator" (Calculator)

Lab: Graphical Application "Summator" (Calculator)

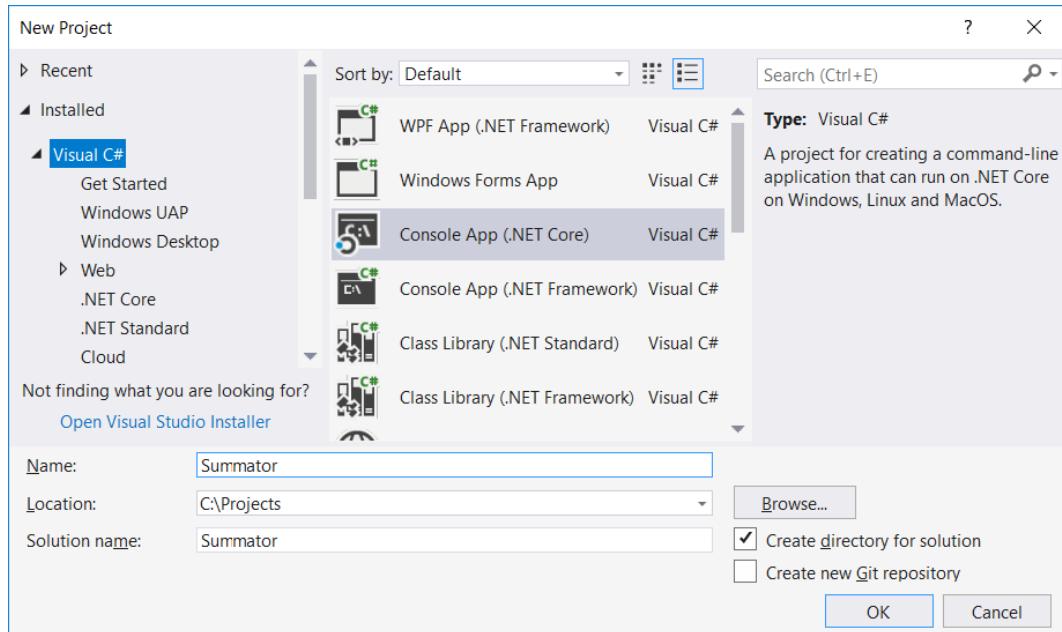
Write a **graphical (GUI) application**, which **calculates the sum of two numbers** (see the screenshot).

By entering two numbers in the first two fields and pressing the button [**Calculate**] their sum is being calculated and the result is shown in the third text field. For our application we will use the **Windows Forms technology**, which allows the creation of **graphical applications for Windows**, in the development environment **Visual Studio** and with programming **language C#**.

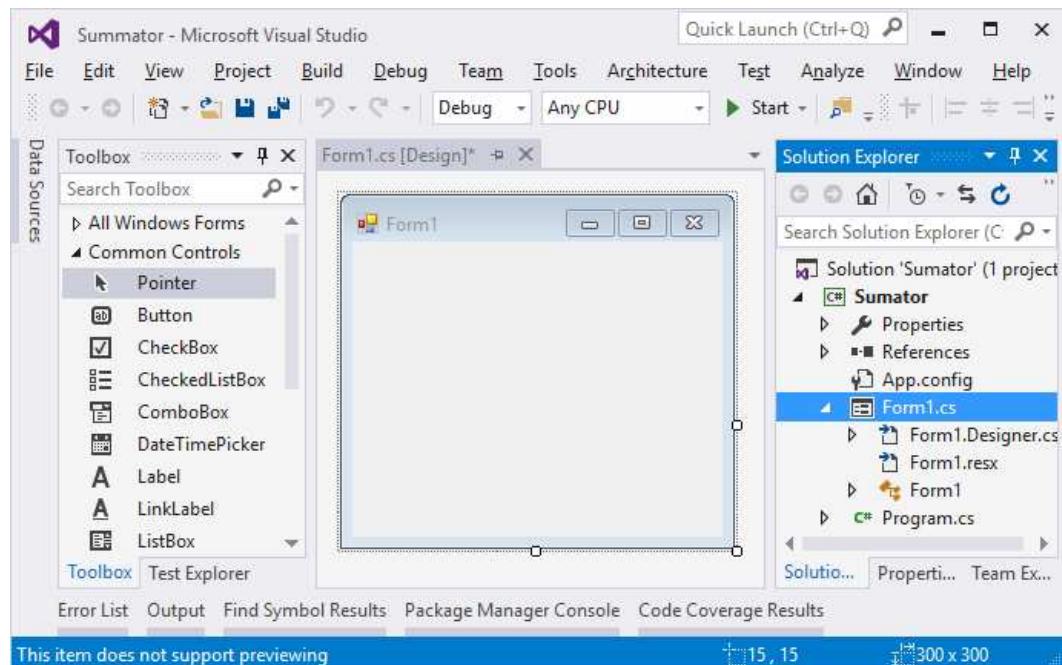


Creating a New C# Project

In Visual Studio we create a new C# project of type “Windows Forms Application”:



When creating a Windows Forms application an editor for user interface will be shown, in which different visual elements could be put (for example text boxes and buttons):

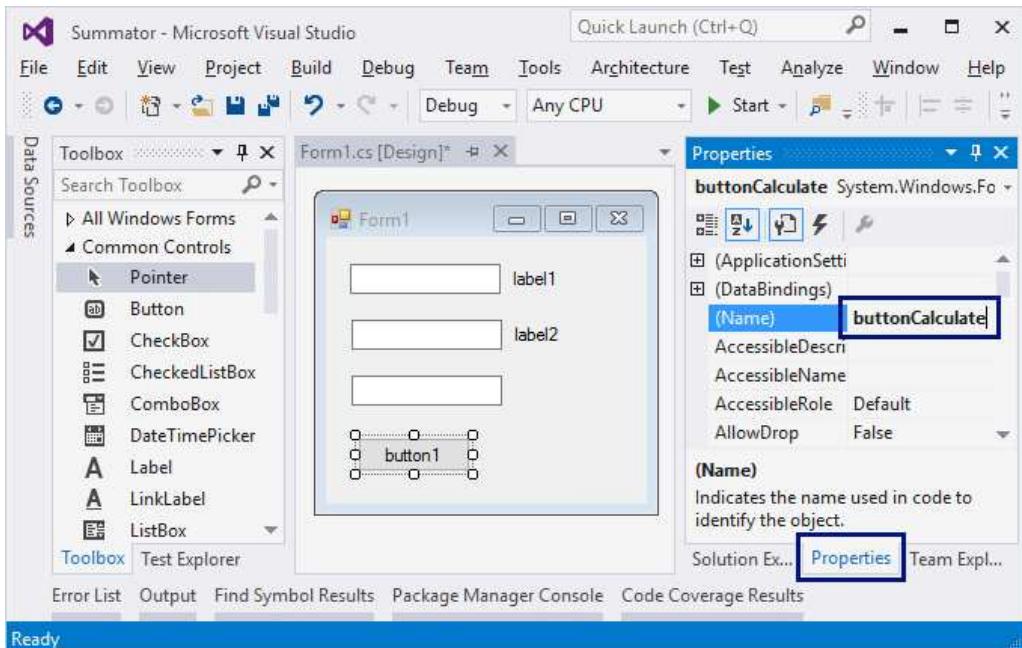


Adding Text Fields and a Button

We drag and drop from the toolbar on the left (Toolbox) three text boxes (TextBox), two labels (Label) and a button (Button), afterwards we arrange them in the window of the application. Then we change the names of each of the controls:

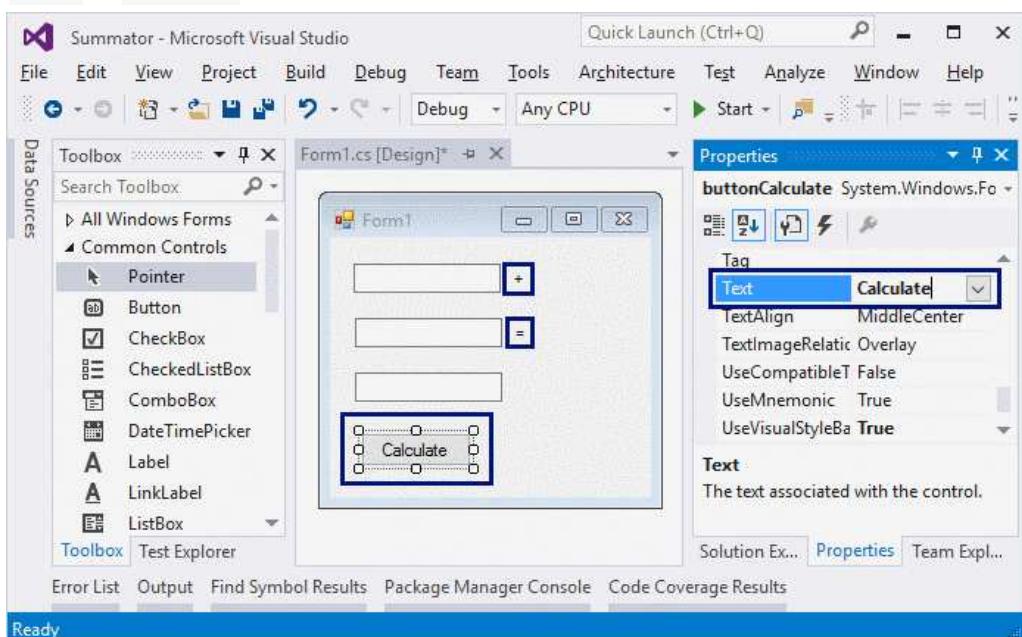
- Names of the text boxes: `textBox1`, `textBox2`, `textBoxSum`
- Name of the button: `buttonCalculate`
- Name of the form: `FormCalculate`

Renaming is done from the window "Properties" on the right, by changing the field (**Name**):



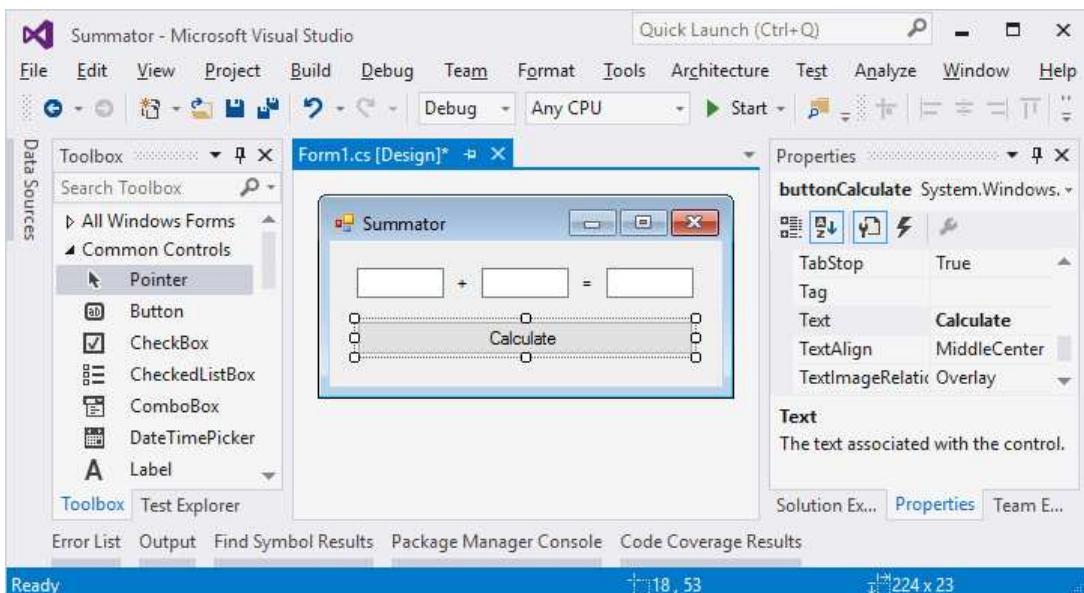
We change the **headings** of the controls (their **Text** property):

- `buttonCalculate` -> `Calculate`
- `label1` -> `+`
- `label2` -> `=`
- `Form1` -> `Summator`

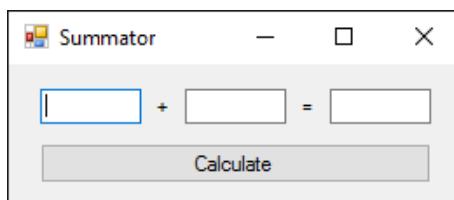


Resizing the Controls and Starting the Application

We resize and arrange the controls, to make them look better:

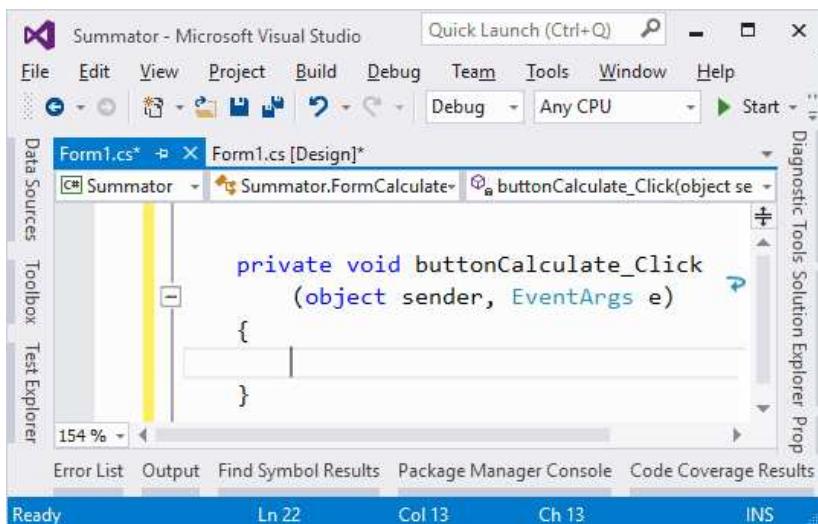


We try to run the application [Ctrl+F5]. It should start, but it should **not** function completely, because we haven't written what happens when we click the button yet.

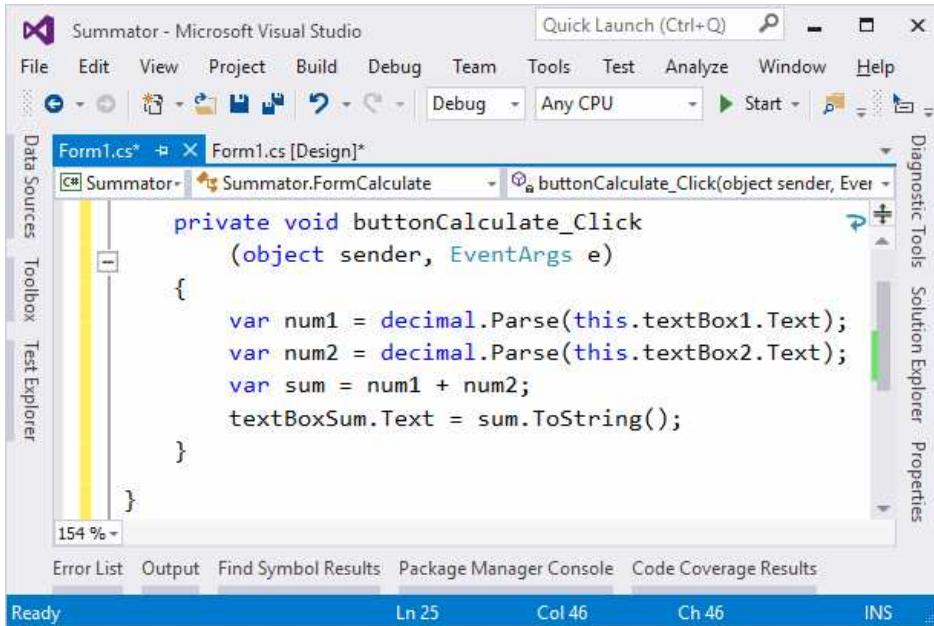


Writing the Program Code

Now it is time to write the code, which **sums the numbers** from the first two fields and **shows the result** in the third field. For this purpose, we double click the **[Calculate]** button. The place, in which we write what is going to happen by clicking the button will be shown:



We write the following C# code between the opening and the closing brackets { }:



```

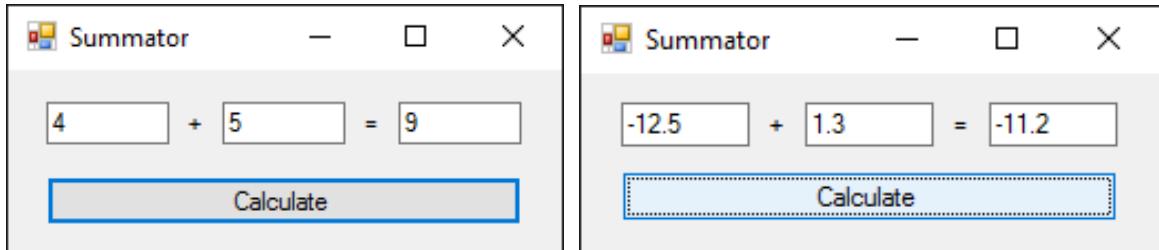
private void buttonCalculate_Click
    (object sender, EventArgs e)
{
    var num1 = decimal.Parse(this.textBox1.Text);
    var num2 = decimal.Parse(this.textBox2.Text);
    var sum = num1 + num2;
    textBoxSum.Text = sum.ToString();
}

```

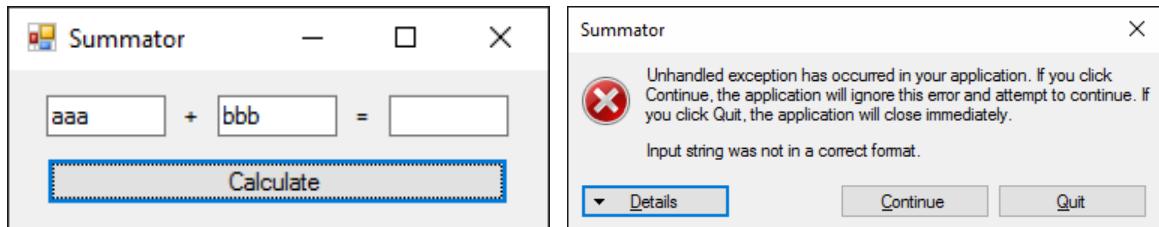
This code takes the first number from the field `textBox1` and keeps it in the variable `num1`, keeps the second number from the field `textBox2` in the variable `num2`, afterwards it sums `num1` and `num2` in the variable `sum` and in the end takes the text value of the variable `sum` in the field `textBoxSum`.

Testing the Application

We start the program again with [Ctrl+F5] and we check whether it works correctly. We try to calculate $4 + 5$, and afterwards $-12.5 + 1.3$:



We try with invalid numbers, for example: "aaa" and "bbb". It seems there is a problem:



Fixing the Bug and Retesting the Application

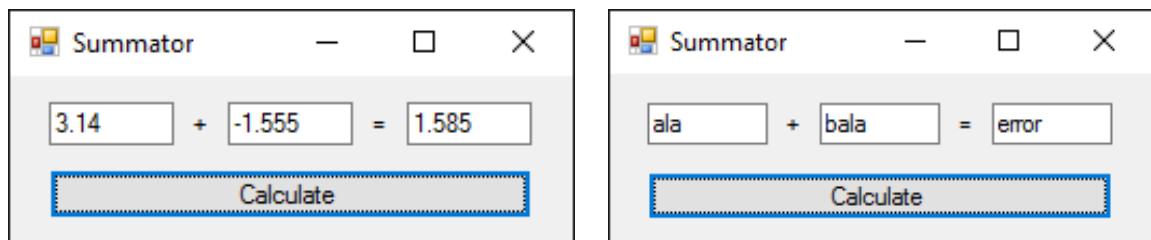
The problem comes from the conversion of the text field into a number. If the value inside the field is not a number, the program throws an exception. We can rewrite the code in order to fix this problem:

```

private void buttonCalculate_Click
    (object sender, EventArgs e)
{
    try
    {
        var num1 = decimal.Parse(this.textBox1.Text);
        var num2 = decimal.Parse(this.textBox2.Text);
        var sum = num1 + num2;
        textBoxSum.Text = sum.ToString();
    }
    catch (Exception)
    {
        textBoxSum.Text = "error";
    }
}

```

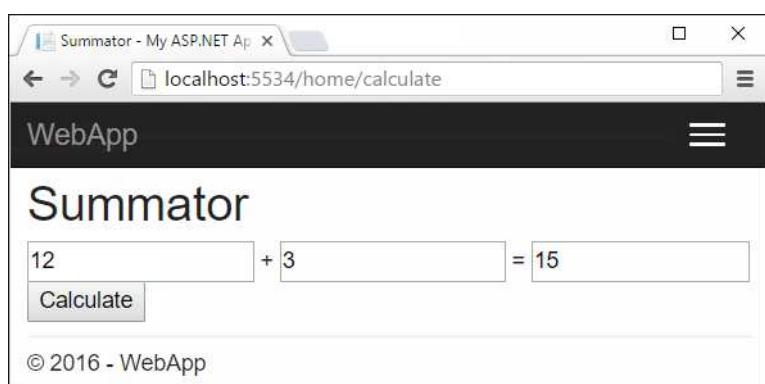
The code above **catches the errors when working with numbers** (it catches exceptions) and in case of an error **it gives a value “error”** in the field holding the result. We start the program again with [Ctrl+F5] and try if it works. This time **by entering a wrong number the result is “error”** and the program continues its normal work:



Is it complicated? It is normal to seem complex, of course. We are just beginning to get into programming. The example above requires much more knowledge and skills, which we are going to develop through this book and even afterwards. Just allow yourself to have some fun with desktop programming. If something doesn't work, you can ask for help in the SoftUni official **discussion forum** (<http://forum.softuni.org>) or in the SoftUni official **Facebook page** (<https://fb.com/softuni.org>). Or move on bravely forward to the next example or to the next chapter of the book. A time will come when it is going to be easy for you, but you really have to put **an effort and be persistent**. Learning programming is a slow process with lots and lots of practice.

Lab: Web Application "Summator" (Calculator)

Now we are going to create something even more complex, but also more interesting: **Web application** that **calculates the sum of two numbers**. By entering two numbers in the first two text fields and pressing the **[Calculate]** button, their sum is calculated, and the result is shown in the third text field. The **Web application** is expected to look similarly to the screenshot.

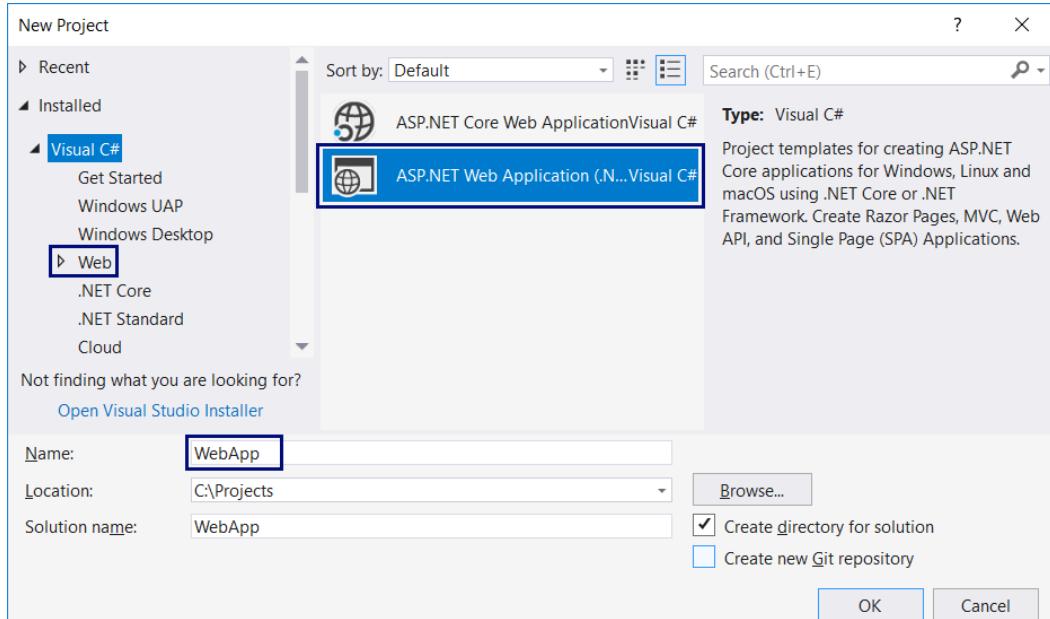


Pay attention that we are creating a **Web-based application**. This is an application that is available through a web browser, just like your favorite email or news website. The web application is going to have a server side (back-end), which is written in the **C#** language with the **ASP.NET MVC** technology, and a client side (front-end), which is written in the **HTML** language (this is a language for visualization of information in a web browser).

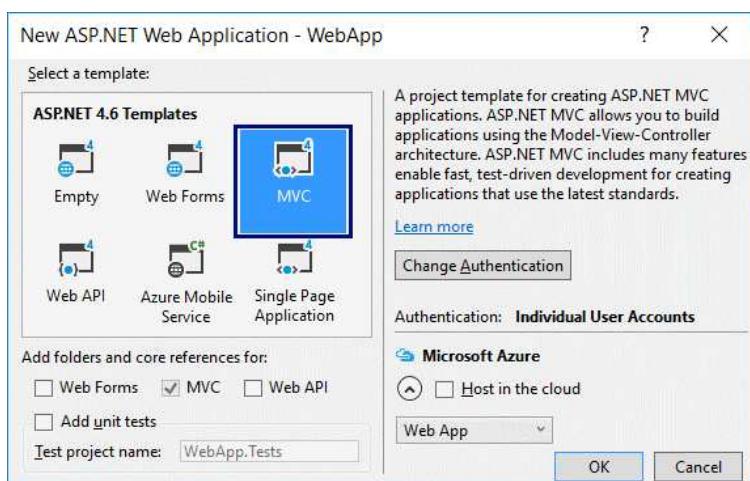
As a difference compared to console applications, which read and write the data in the form of a text on the console, Web applications have a **Web-based user interface**. Web applications **are being loaded from some Internet address** (URL) through a standard web browser. Users write input data in a page, visualized from the web browser, the data is processed on a web server and the results are shown again in a page of the web browser. For our web application we are going to use the **ASP.NET MVC technology**, which allows creating of **web applications with the programming language C#** in the development environment **Visual Studio**.

Creating a New ASP.NET MVC Project

In Visual Studio we create a new C# project of type “**ASP.NET Web Application**”, named **WebApp**:

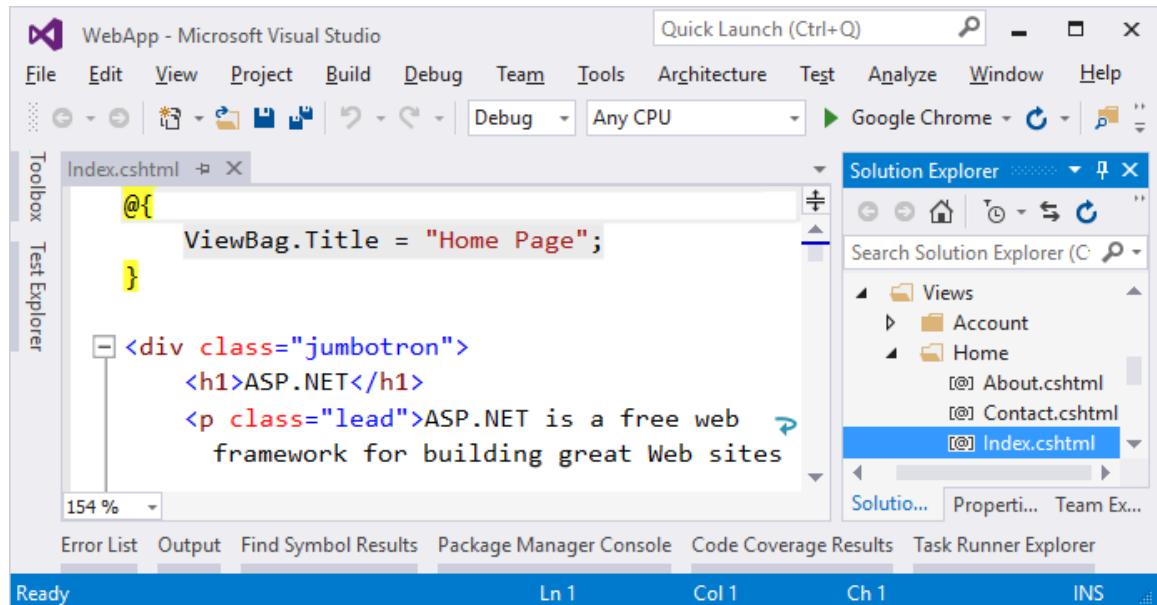


We choose as **type** of the application: “**MVC**”:



Creating a View (Web Form)

We find the file `Views\Home\Index.cshtml`. The view of the home page of our web app is inside it:



We delete the old code from the file `Index.cshtml` and write the following code:

```

@{
    ViewBag.Title = "Summator";
}

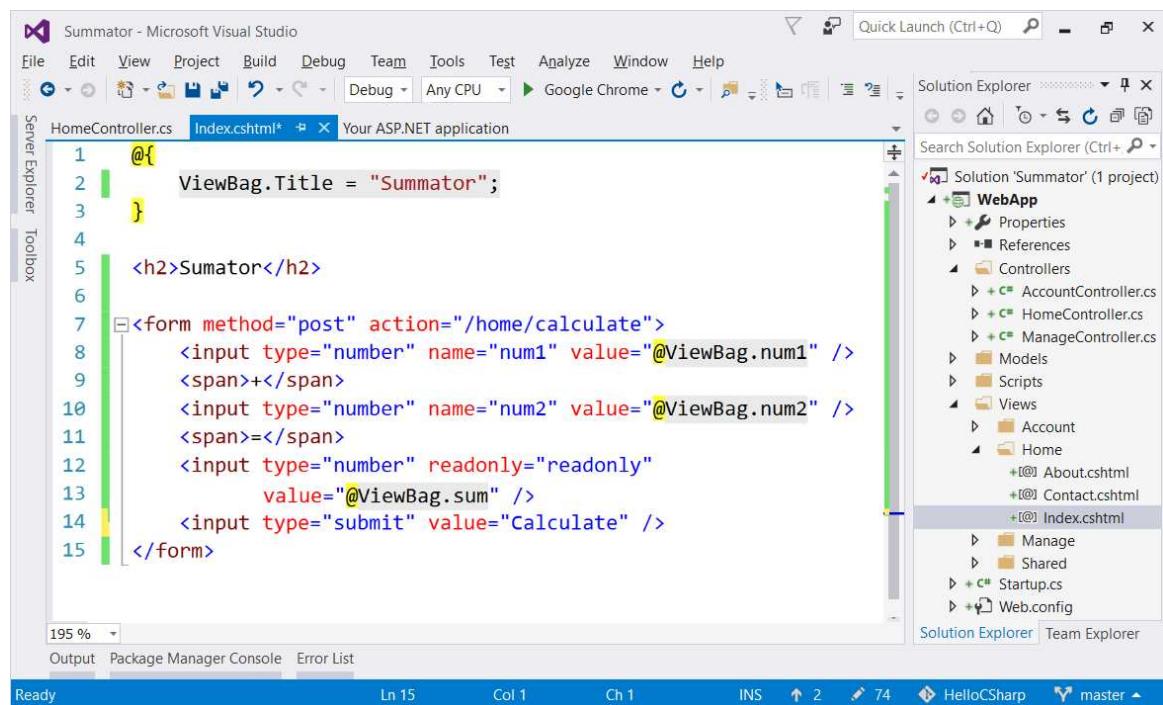
<h2>Summator</h2>

<form method="post" action="/home/calculate">
    <input type="number" name="num1" value="@ViewBag.num1" />
    <span>+</span>
    <input type="number" name="num2" value="@ViewBag.num2" />
    <span>=</span>
    <input type="number" readonly="readonly" value="@ViewBag.sum" />
    <input type="submit" value="Calculate" />
</form>

```

This code creates a web form with three text boxes and a button in it. Inside the fields, values are being loaded, which are calculated previously in the object `ViewBag`. The requirement says that with the click of the [Calculate] button the action `/home/calculate` (action `calculate` from the `home controller`) will be called.

This is how the file `Index.cshtml` is supposed to look after the change:



Writing the Program Code

What remains is to write the action that sums the numbers when clicking the button [Calculate]. We open the file **Controllers\HomeController.cs** and we add the following code into the body of **HomeController** class:

```
public ActionResult Calculate(int num1, int num2)
{
    this.ViewBag.num1 = num1;
    this.ViewBag.num2 = num2;
    this.ViewBag.sum = num1 + num2;
    return View("Index");
}
```

This code implements the “calculate” action. It takes two parameters **num1** and **num2** and stores them in the **ViewBag** object, after which it **calculates and stores their sum** as well. The values stored in **ViewBag** are later **used from the view**. These values are shown in the **three text fields** inside the form for summing numbers in the web page of the application.

Here is how **the file HomeController.cs** should look after the change:

The screenshot shows the Microsoft Visual Studio interface. The code editor on the left displays the `HomeController.cs` file with the following code:

```

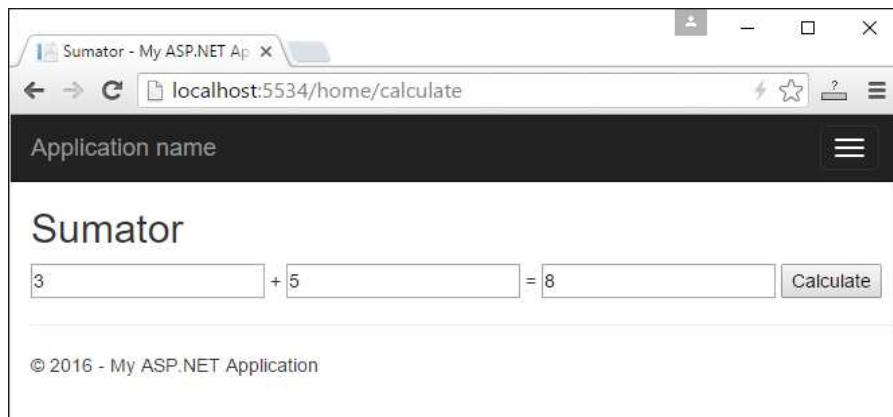
4 {
5     0 references | 0 requests
6     public class HomeController : Controller
7     {
8         0 references | 0 requests | 0 exceptions
9         public ActionResult Index()
10        {
11            return View();
12        }
13
14        0 references | 0 requests | 0 exceptions
15        public ActionResult Calculate(int num1, int num2)
16        {
17            this.ViewBag.num1 = num1;
18            this.ViewBag.num2 = num2;
19            this.ViewBag.sum = num1 + num2;
20
21            return View("Index");
22        }
23    }
24 }

```

The Solution Explorer on the right shows the project structure for "WebApp". The `HomeController.cs` file is selected.

Testing the Web Application

The application is ready. We can start it with [Ctrl+F5] and test whether it works:



Does it look scary? **Don't be afraid!** We have a lot more to learn, to reach the level of knowledge and skills to write web-based applications freely like in the example above, as well as much bigger and much more complex ones. If you don't succeed, there is nothing to worry about, keep moving on. After some time, you will remember with a smile how incomprehensible and exiting your first collision with web programming was. If you have problems with the example above, you can ask for help in the SoftUni official [discussion forum](#) (<http://forum.softuni.org>) or in the SoftUni official [Facebook page](#) (<https://fb.com/softuni.org>).

The purpose of both of the above examples (graphical desktop application and web application) is not to teach you, but to make you dive a little deeper into programming, **to enhance your interest** towards software development and **to inspire you to learn** hard.

You have a lot to learn yet, but it's interesting, isn't it?

Chapter 2.1. Simple Calculations

In this chapter we are going to get familiar with the following concepts and programming techniques:

- What is the **system console**?
- How to **read numbers** from the system console?
- How to work with **data types and variables**, which are necessary to process numbers and the operations between them?
- How to **print** output (a number) on the console?
- How to do simple **arithmetic operations**: add, subtract, multiply, divide, string concatenation?

Video: Chapter Overview

Watch a video about what shall we learn in this chapter here: https://youtu.be/NXbFJw_NstA.

Introduction to Simple Calculations by Examples

Computer programs can **enter data** from the **console**, perform **calculations** and **print** the results on the console. This is a simple example of C# program that **converts** from **feet** to **meters**:

```
Console.Write("Feet = ");
var feet = double.Parse(Console.ReadLine());
var meters = feet * 0.3048;
Console.WriteLine("Meters = ");
Console.WriteLine(meters);
```

Run the above code example: <https://repl.it/@nakov/feet-to-meters-csharp>.

The above program **enters a number** and **converts** its value from **feet** to **meters**. This is a **sample output** from the above code, when the user enters **5** as input:

```
Feet = 5
Meters = 1.524
```

In C# we can **read a text line** from the console using **Console.ReadLine()** and we can convert the text to a floating-point number using **double.Parse(text)**. We can **print text and numbers** using the **\$ text formatting** syntax as follows:

```
var radius = 1.25;
Console.WriteLine($"Circle radius = {radius}");
Console.WriteLine($"Circle area = {Math.PI * radius * radius}");
```

Run the above code example: <https://repl.it/@nakov/circle-area-csharp>.

The **\$ syntax** replaces all expressions in curly brackets with their values. The output from the above code is:

```
Circle radius = 1.25
Circle area = 4.90873852123405
```

Let's explain in greater detail how to use the **console**, how to **enter numbers** and **text** and how to **perform simple calculations** and **format and print text** and expressions on the console in C#.

The System Console

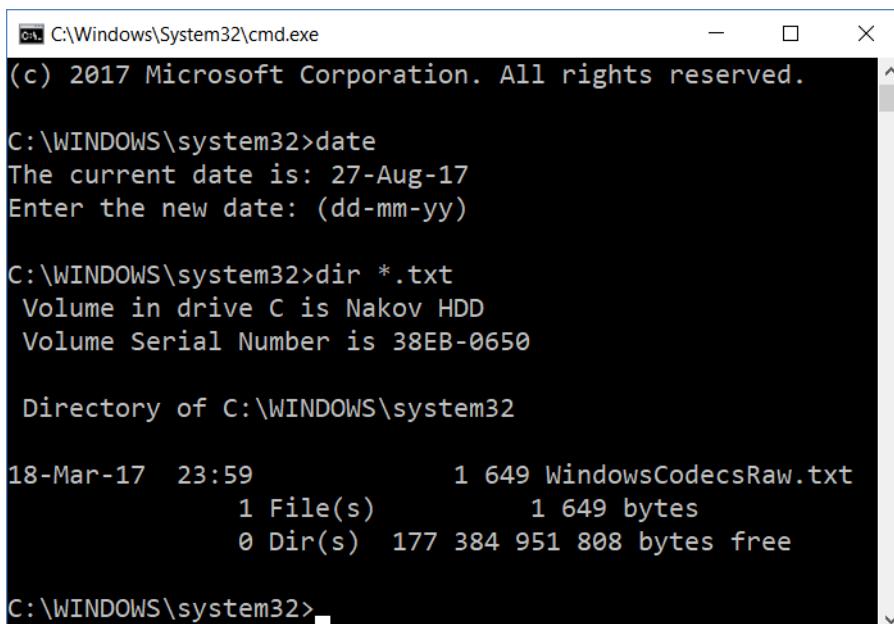
Simply called "console", the "system console", the "system terminal", the "standard input / output", also the "computer command line console", represents the tool by which we give the computer commands in a text format and get the results from their execution again as a text.

Video: The System Console

Watch a video lesson about the system console here: <https://youtu.be/ehHnNu6M55M>.

The System Console Explained

Generally, the **system console** represents a text **terminal**, which means that it accepts and visualizes just **text** without any graphical elements like buttons, menus, etc. It usually looks like a black colored window like this one:



```
C:\Windows\System32\cmd.exe
(c) 2017 Microsoft Corporation. All rights reserved.

C:\WINDOWS\system32>date
The current date is: 27-Aug-17
Enter the new date: (dd-mm-yy)

C:\WINDOWS\system32>dir *.txt
Volume in drive C is Nakov HDD
Volume Serial Number is 38EB-0650

Directory of C:\WINDOWS\system32

18-Mar-17 23:59           1 649 WindowsCodecsRaw.txt
      1 File(s)            1 649 bytes
      0 Dir(s) 177 384 951 808 bytes free

C:\WINDOWS\system32>
```

In most operating systems, the **console** is available as a standalone application on which we write console commands. It is called a **Command Prompt** in Windows, and a **Terminal** in Linux and Mac. The console runs console applications. They read text from the command line and print text on the console. In this book we are going to learn programming mostly through creating **console applications**.

In the next examples we will **read data** (like integers, floating-point numbers and strings) from the console and will **print data** on the console (text and numbers).

Reading Integers from the Console

In order to read an **integer** (not a float) **number** from the console, we have to **declare a variable**, declare the **number type** and use the standard command for **reading a text line** from the system console **Console.ReadLine()** and after that **convert the text line into an integer number** using **int.Parse(text)**:

```
var num = int.Parse(Console.ReadLine());
```

The above line of C# code **reads an integer** from the first line on the console.

Video: Reading Data from the Console

Watch a video lesson about reading from the system console: <https://youtu.be/WPIQ5HYBGJQ>.

Video: Reading Integers from the Console

Watch a video lesson about reading integers from the console: <https://youtu.be/3TC2F-ffw34>.

Example: Calculating a Square Area

For example, let us look at the following program, which **reads an integer from the console**, multiplies it by itself (**squares** it) and **prints the result** from the multiplication.

Video: Calculating a Square Area

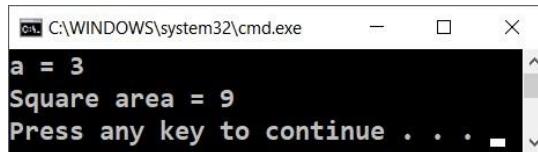
Watch a video lesson about calculating square area: <https://youtu.be/gdYTotTFVgA>.

Code: Calculating a Square Area

This code demonstrates how we can calculate the **square area** by the given length of the side:

```
Console.Write("a = ");
var a = int.Parse(Console.ReadLine());
var area = a * a;
Console.Write("Square area = ");
Console.WriteLine(area);
```

Here is how the program would work when we have a square with a side length equal to 3:



Try to write a wrong number, for example "hello". You will get an error message during runtime (exception). This is normal. Later on, we will find out how we can catch these kinds of errors and make the user enter a number again.

Testing in the Judge System

Test your solution here: <https://judge.softuni.org/Contests/Practice/Index/504#0>.

How Does the Example Work?

The first line `Console.Write("a = ");` prints an informative message, which invites the user to enter the side of the square `a`. After the output is printed, the cursor stays on the same line. Staying on the same line is more convenient for the user, visually. We use `Console.Write(...)`, and not `Console.WriteLine(...)` and this way the cursor stays on the same line.

The next line `var a = int.Parse(Console.ReadLine());` reads an integer from the console. Actually, it first reads a text (string) using `Console.ReadLine()` and after that it gets converted to an integer (it is parsed) using `int.Parse(...)`. The result is kept in a variable with name `a`.

The next command `var area = a * a;` keeps in a new variable `area` the result of the multiplication of `a` by `a`.

The next command `Console.WriteLine("Square area = ")`; prints the given text without going to the next line. Again, use `Console.WriteLine(...)`, and not `Console.WriteLine(...)`, and this way the cursor stays on the same line in order to print the calculated area of the square afterwards.

The last command `Console.WriteLine(area);` prints the calculated value of the variable `area`.

Data Types and Variables

In programming, each variable stores a certain **value** of a particular **type**. For example, data types can be number, letter, text (string), date, color, image, list, etc. Here are some examples of data types:

- **integer**: 1, 2, 3, 4, 5, 20, ...
- **float**: 0.5, 3.14, -1.5, ...
- **character** (symbol): 'a', 'b', 'c', '@', 'X', ...
- **text** (string): "Hello", "Hi", "How are you?", ...
- **day of week**: Monday, Tuesday, ..., Sunday
- **date and time**: 14-June-1980 6:30:00, 25-Dec-2017 23:17:22

Video: Data Types and Variables

Watch a video lesson about declaring variables: <https://youtu.be/p4tedmW8dyw>.

Examples: Data Types and Variables

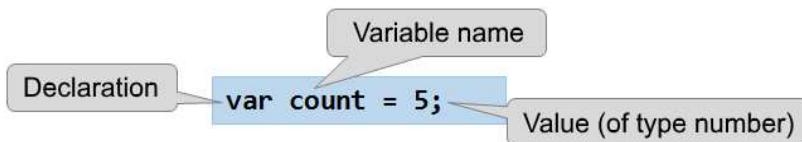
In C# we can use **data types** to define **variables** as follows:

```
int a = 5;
string str = "Some text";
char letter = 'A';
float f = 4.2;
```

In C#, once a **variable** is defined, it can **change its value** many times, but it **cannot change its data type** later. Variables may hold only data of their type.

Declaring and Using Variables

We know that computers are machines that process data. All **data** is stored inside the computer memory (RAM) in **variables**. The variables are named areas in the memory, which keep a certain data type, for example a number or a text. Each of the **variables** in C# has a **name**, a **type** and a **value**. Here is how we would **declare a variable** and **assign it** with a **value** at the same time:



Video: Declaring and using Variables

Watch a video lesson about declaring variables: <https://youtu.be/g-dG5GobHg0>.

Examples: Declaring and using Variables

Example of declaring a variable:

```
var count = 5;
```

After being processed, data is again **stored in variables** (in some place in the memory allocated for our program):

```
count = count + 1;
```

After the above code the variable **count** changes its value and increases by **1**.

Reading Floating Point Numbers from the Console

To read a **floating-point number** (fractional number, non-integer) from the console use the following command:

```
var num = double.Parse(Console.ReadLine());
```

The above C# code first reads a **text line** from the console, then converts (parses) it to a **floating-point number**.

Video: Reading Floating-Point Numbers

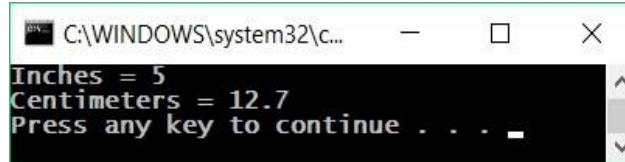
Watch the following video lesson about how to read floating-point numbers from the console: <https://youtu.be/H2waLeIW70A>.

Example: Converting Inches into Centimeters

Let's write a program that reads a floating-point number in inches and converts it to centimeters:

```
Console.Write("Inches = ");
var inches = double.Parse(Console.ReadLine());
var centimeters = inches * 2.54;
Console.Write("Centimeters = ");
Console.WriteLine(centimeters);
```

Let's start the program and make sure that when a value in inches is entered, we obtain a correct output in centimeters:



Note that if you enter an invalid number, e.g. "asfd", the program will crash with an error message (exception). We will learn how to handle exceptions later in the chapter "[More Complex Loops](#)".

Testing in the Judge System

Test your solution here: <https://judge.softuni.org/Contests/Practice/Index/504#1>.

Reading a Text from the Console

To read a **text** (string) from the console, again, we have to **declare a new variable** and use the standard **command for reading a text from the console**:

```
var str = Console.ReadLine();
```

By default, the `Console.ReadLine(...)` method returns a **text result** – a text line, read from the console.

- After you read a text from the console, additionally, you can **parse the text** to an integer by `int.Parse(...)` or a floating-point number by `double.Parse(...)`.
- If parsing to a number is not done, **each number** will simply be **text**, and we **cannot do** arithmetic operations with it.

Video: Reading Text from the Console

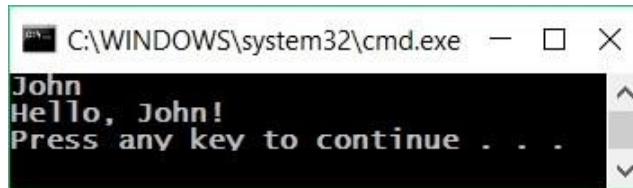
Watch a video lesson about how to read text from the console: <https://youtu.be/0tzvEdWxZ1k>.

Example: Greeting by Name

Let's write a program that asks the user for their **name** and salutes them with the text "Hello, <name>!".

```
var name = Console.ReadLine();
Console.WriteLine("Hello, {0}!", name);
```

In this case the `{0}` expression is replaced with the **first** passed argument, which holds the variable **name**. If we enter "John", the output will be as follows:



Testing in the Judge System

Test your solution here: <https://judge.softuni.org/Contests/Practice/Index/504#2>.

Printing and Formatting Text and Numbers

In C#, when printing a text, numbers and other data on the console, we **can join them** by using templates `{0}`, `{1}`, `{2}` etc. In programming, these templates are called **placeholders**. This is a simple example:

```
Console.WriteLine("{0} + {1} = {2}", 3, 5, 3+5);
```

The placeholders `{0}`, `{1}` and `{2}` are replaced by the expressions, given after the text. The result from the above code is:

```
3 + 5 = 8
```

Video: Printing Text and Numbers

Watch a video lesson about how to print text and numbers together on the console: <https://youtu.be/tSTwvaQpy9g>.

Example: Printing Text and Numbers

```
var firstName = Console.ReadLine();
var lastName = Console.ReadLine();
var age = int.Parse(Console.ReadLine());
var town = Console.ReadLine();
Console.WriteLine("You are {0} {1}, a {2}-years old person from {3}.",
    firstName, lastName, age, town);
```

This is the **result** we are going to obtain after the execution of this example:

```
Ivan
Ivanov
30
Plovdiv
You are Ivan Ivanov, a 30-years old person from Plovdiv.
Press any key to continue . . .
```

Notice how every variable should be passed in the **order**, in which we want it to be printed. Practically, the template (**placeholder**) accepts variables of any type.

It is possible for a template to be used **multiple times** and it is not necessary for the templates to be numbered sequentially. Here is an **example**:

```
Console.WriteLine("{1} + {1} = {0}", 1+1, 1);
```

The result is:

```
1 + 1 = 2
```

Testing in the Judge System

Test your solution here: <https://judge.softuni.org/Contests/Practice/Index/504#3>.

Using the Dollar String Interpolation

We can format text in C# using also the following **\$ syntax**. It provides simplifies text formatting:

```
var a = 4.5;
Console.WriteLine($"Square size = {a}");
Console.WriteLine($"Square area = {a * a}");
```

The output from the above code is as follows:

```
Square size = 4.5
Square area = 20.25
```

The **\$** prefix before a string in C# enables the so called "**string interpolation**": replacing all expressions, staying in curly brackets **{ }** in the text with their values.

Using the **dollar string interpolation syntax**, the last example can be rewritten like this:

```
var firstName = Console.ReadLine();
var lastName = Console.ReadLine();
```

```
var age = int.Parse(Console.ReadLine());
var town = Console.ReadLine();
Console.WriteLine($"You are {firstName} {lastName}, a {age}-years old
person from {town}.");
```

Play with the above code and test it in the SoftUni online Judge system: <https://judge.softuni.org/Contests/Practice/Index/504#3>.

Arithmetic Operations

Let's examine the basic **arithmetic operations** in programming. We can add, subtract, multiply and divide numbers using the operators `+`, `-`, `*` and `/`.

Video: Arithmetic Operators

Watch a video lesson about the arithmetic operators: <https://youtu.be/XOtEuEUbA4M>.

Summing up Numbers: Operator `+`

We can **sum** up numbers using the `+` operator:

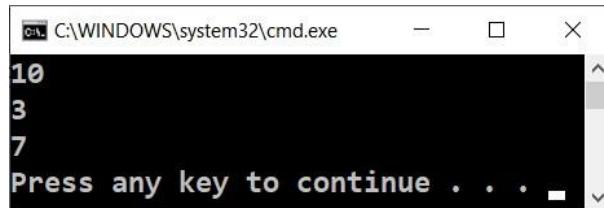
```
var a = 5;
var b = 7;
var sum = a + b; // the result is 12
```

Subtracting Numbers: Operator `-`

Subtracting numbers is done using the `-` operator:

```
var a = int.Parse(Console.ReadLine());
var b = int.Parse(Console.ReadLine());
var result = a - b;
Console.WriteLine(result);
```

Here is the result of the execution of this program (with numbers 10 and 3):



Multiplying Numbers: Operator `*`

For **multiplication** of numbers we use the `*` operator:

```
var a = 5;
var b = 7;
var product = a * b; // 35
```

Dividing Numbers: Operator `/`

Dividing numbers is done using the `/` operator. It works differently with **integers** and **floating-point numbers**.

- When we divide two integers, an **integer division** is applied, and the obtained output is without its fractional part. Example: $11 / 3 = 3$.
- When we divide two numbers and at least one of them is a float number, a **floating division** is applied, and the obtained result is a float number, just like in math. Example $11 / 4.0 = 2.75$. When it cannot be done with exact precision, the result is being rounded, for example $11.0 / 3 = 3.66666666666667$.
- The integer **division by 0** causes an **exception** during runtime (runtime exception).
- Float numbers **divided by 0** do not cause an exception and the result is **$+\text{- infinity}$** or a special value **NaN**. Example $5 / 0.0 = \infty$.

Here are a few **examples** with the division operator:

```
var a = 25;
var i = a / 4;           // we are applying an integer division:
                        // the result of this operation will be 6 -
                        // the fractional part will be cut,
                        // because we are dividing integers
var f = a / 4.0;        // 6.25 - floating division. We have set the
                        // number 4 to be interpreted
                        // as a float by adding a decimal separator
                        // followed by zero
var error = a / 0;      // Error: Integer divided by zero
```

Dividing Integers

Let's examine a few examples for **integer division** (remember that when we **divide integers** in C# the result is an **integer**):

```
var a = 25;
Console.WriteLine(a / 4); // Integer result: 6
Console.WriteLine(a / 0); // Error: divide by 0
```

Dividing Floating-Point Numbers

Let's look at a few examples for **floating division**. When we divide floating point numbers, the result is always a **float number** and the division never fails, and works correctly with the special values **$+\infty$** and **$-\infty$** :

```
var a = 15;
Console.WriteLine(a / 2.0);   // Float result: 7.5
Console.WriteLine(a / 0.0);   // Result: Infinity
Console.WriteLine(-a / 0.0);  // Result: -Infinity
Console.WriteLine(0.0 / 0.0); // Result: NaN (Not a Number), e.g. the
                           // result from
                           // the operation is not a valid numeric value
```

When printing the values **∞** and **$-\infty$** , the console output may be **?**, because the console in Windows does not work correctly with Unicode and breaks most of the non-standard symbols, letters and special characters. The example above would most probably give the following result:

```
7.5
?
-
NaN
```

Concatenating Text and Numbers

Besides for summing up numbers, the operator `+` is also used for **joining pieces of text** (concatenation of two strings one after another). In programming, joining two pieces of text is called "**concatenation**". Here is how we can concatenate a text with a number by the `+` operator:

```
var firstName = "Maria";
var lastName = "Ivanova";
var age = 19;
var str = firstName + " " + lastName + " @ " + age;
Console.WriteLine(str); // Maria Ivanova @ 19
```

Video: Concatenating Text and Numbers

Watch a video lesson about concatenating text and numbers: https://youtu.be/vPI-V2NG_CU.

Examples: Concatenating Text and Numbers

Here is another **example** of concatenating text and numbers:

```
var a = 1.5;
var b = 2.5;
var sum = "The sum is: " + a + b;
Console.WriteLine(sum); // The sum is: 1.52.5
```

Did you notice **something strange**? Maybe you expected the numbers **a** and **b** to be summed? Actually, the concatenation works from right to left and the result above is absolutely correct.

If we want to sum the numbers, we have to use **brackets**, in order to change the order of execution of the operations:

```
var a = 1.5;
var b = 2.5;
var sum = "The sum is: " + (a + b);
Console.WriteLine(sum); // The sum is: 4
```

Numerical Expressions

In programming, we can calculate **numerical expressions**, for example:

```
var expr = (3 + 5) * (4 - 2);
```

The standard rule for priorities of arithmetic operations is applied: **multiplying and dividing are always done before adding and subtracting**. In case of an **expression in brackets**, it is calculated first, but we already know all of that from school math.

Video: Numerical Expressions

Watch a video lesson about numerical expressions: <https://youtu.be/6MPxIOCsPdw>.

Example: Calculating Trapezoid Area

Let's write a program that inputs the lengths of the two bases of a trapezoid and its height (one floating point number per line) and calculates the **area of the trapezoid** by the standard math formula:

```
var b1 = double.Parse(Console.ReadLine());
var b2 = double.Parse(Console.ReadLine());
var h = double.Parse(Console.ReadLine());
var area = (b1 + b2) * h / 2.0;
Console.WriteLine("Trapezoid area = " + area);
```

If we start the program and enter values for the sides: 3, 4 and 5, we will obtain the following result:

```
3
4
5
Trapezoid area = 17.5
```

Testing in the Judge System

Test your solution here: <https://judge.softuni.org/Contests/Practice/Index/504#4>.

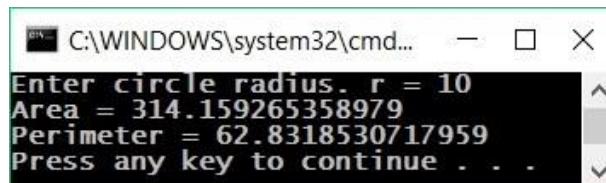
Example: Circle Area and Perimeter

Let's write a program that calculates **a circle area and perimeter** by reading its **radius r**. Formulas:

- Area = $\pi * r * r$
- Perimeter = $2 * \pi * r$
- $\pi \approx 3.14159265358979323846\ldots$

```
Console.WriteLine("Enter circle radius. r = ");
var r = double.Parse(Console.ReadLine());
Console.WriteLine("Area = " + Math.PI * r * r);
// Math.PI - built-in constant for π in C#
Console.WriteLine("Perimeter = " + 2 * Math.PI * r);
```

Let's test the program with **radius r = 10**:



Testing in the Judge System

Test your solution here: <https://judge.softuni.org/Contests/Practice/Index/504#5>.

Example: 2D Rectangle Area

A rectangle is given with the coordinates of two of its opposite angles. Calculate its area and perimeter (see the screenshot).

In this task, we have to consider that if we subtract the smaller x from the bigger x , we will obtain the length of the rectangle. Identically, if we subtract the smaller y from the bigger y , we will obtain the height of the rectangle. What is left is to multiply both sides. Here is an example of an implementation of the described logic:

```
var x1 = double.Parse(Console.ReadLine());
var y1 = double.Parse(Console.ReadLine());
var x2 = double.Parse(Console.ReadLine());
var y2 = double.Parse(Console.ReadLine());

// Calculating the sides of the rectangle:
var width = Math.Max(x1, x2) - Math.Min(x1, x2);
var height = Math.Max(y1, y2) - Math.Min(y1, y2);

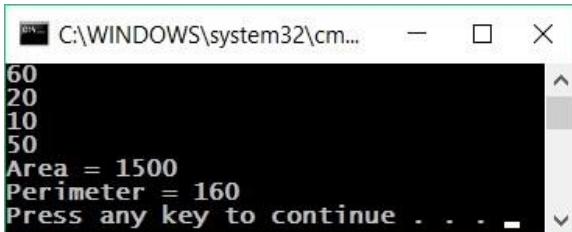
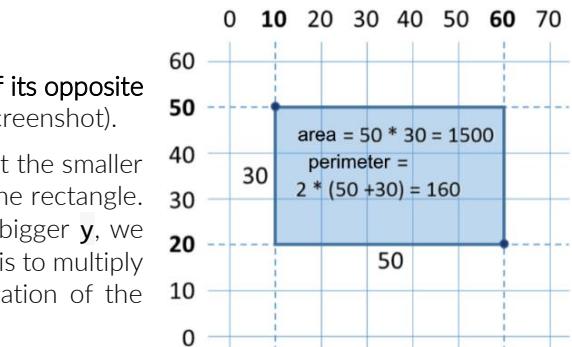
Console.WriteLine("Area = " + width * height);
Console.WriteLine("Perimeter = " + 2 * (width + height));
```

We use `Math.Max(a, b)`, to find the higher value from a and b and identically `Math.Min(a, b)` to find the lower of both values.

When the program is executed with the values from the coordinate system given in the above example, we obtain the result, shown at the screenshot.

Testing in the Judge System

Test your solution here: <https://judge.softuni.org/Contests/Practice/Index/504#6>.



Other Expressions

Expressions in C# can be not only numerical, but also can be text expressions, date expressions or expressions of other type:

```
var price = 20;
var priceUSD = "$ " + price; // text expression
var priceGBP = price + " GBP";
Console.WriteLine(priceUSD); // $ 20
Console.WriteLine(priceGBP); // 20 GBP

var date = new DateTime(2017, 6, 14);
var dateAfter5days = date.AddDays(5); // 14-Jun-17 (date expression)
Console.WriteLine(dateAfter5days); // 19-Jun-17 00:00:00
```

Exercises: Simple Calculations

Let's strengthen the knowledge gained throughout this chapter with a few [more exercises](#).

Video: Chapter Summary

Watch the following video to summarize what we learned in this chapter about working with simple calculations: https://youtu.be/Zv_c-M_7Gyw.

What We Learned in This Chapter?

Let's summarize what we learned in this chapter:

- Reading a text: `var str = Console.ReadLine();`
- Reading an integer: `var num = int.Parse(Console.ReadLine());`
- Reading a floating-point number: `var num = double.Parse(Console.ReadLine());`
- Calculations with numbers and using the arithmetic operators +, -, *, /, (): `var sum = 5 + 3;`
- Printing a text by placeholders on the console: `Console.WriteLine("{0} + {1} = {2}", 3, 5, 3 + 5);`

The Exercises

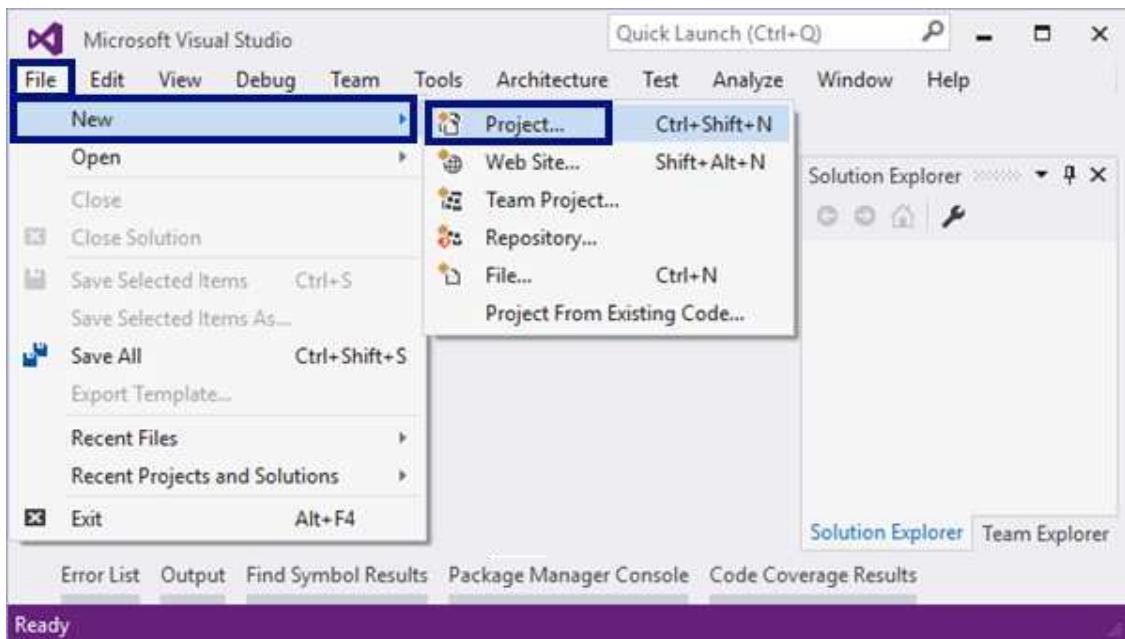
We have a lot of **practical work**. Solve the exercises at the end of this chapter to learn how to work with variables and data types, reading and writing on the console, using data and calculations.

Empty (Blank) Visual Studio Solution

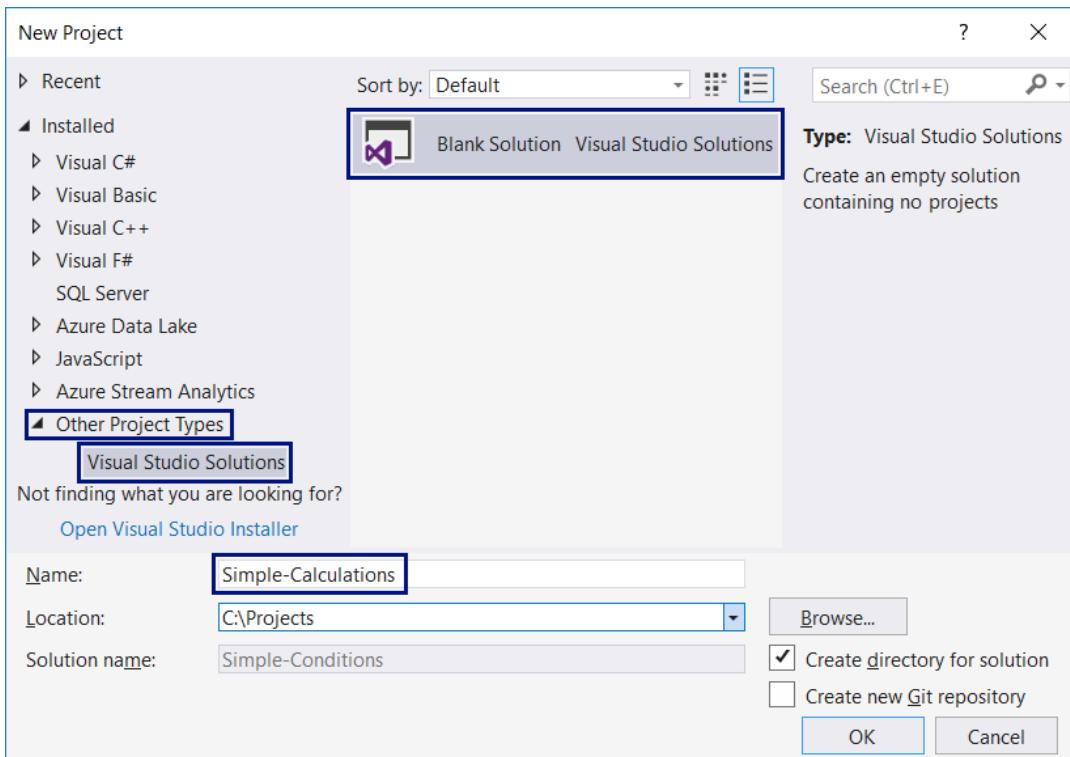
We start by creating an empty solution (**Blank Solution**) in Visual Studio. The solutions in Visual Studio combine **a group of projects**. This opportunity is **very convenient**, when we want to **work on a few projects** and switch quickly between them or we want to **unite logically a few interconnected projects**.

In the current practical exercise, we will use a "**Blank Solution**" with **a couple of projects** to organize the solutions of the exercises – every task in a separate project and all of them in a common solution.

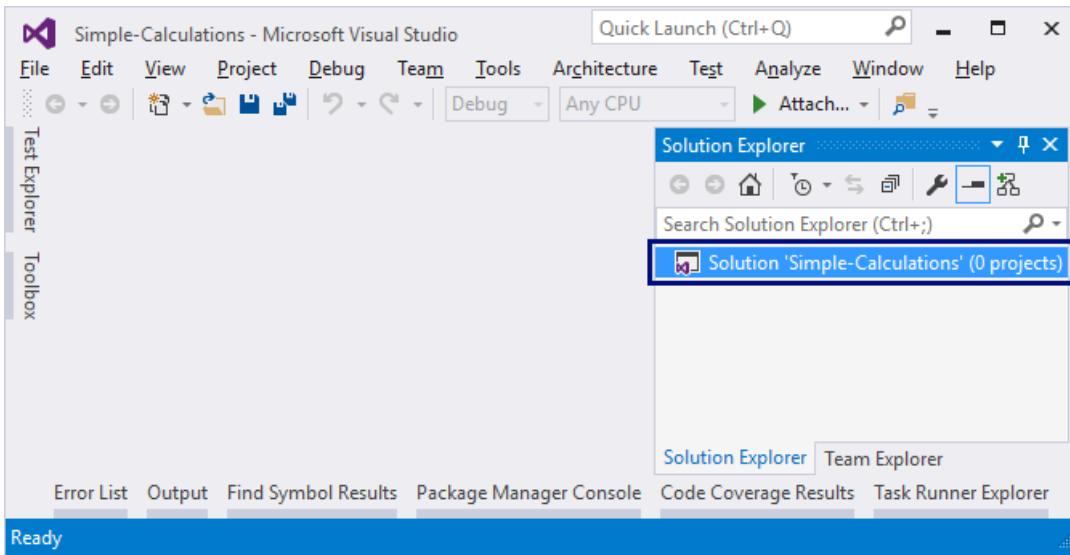
- We start Visual Studio
- We create a new **Blank Solution**: [File] -> [New] -> [Project].



We choose from the **templates** -> [Other Project Types] -> [Visual Studio Solutions] -> [Blank Solution] and we give an appropriate name of the project, for example “Simple-Calculations”:



Now we have created an **empty Visual Studio Solution** (with 0 projects in it):



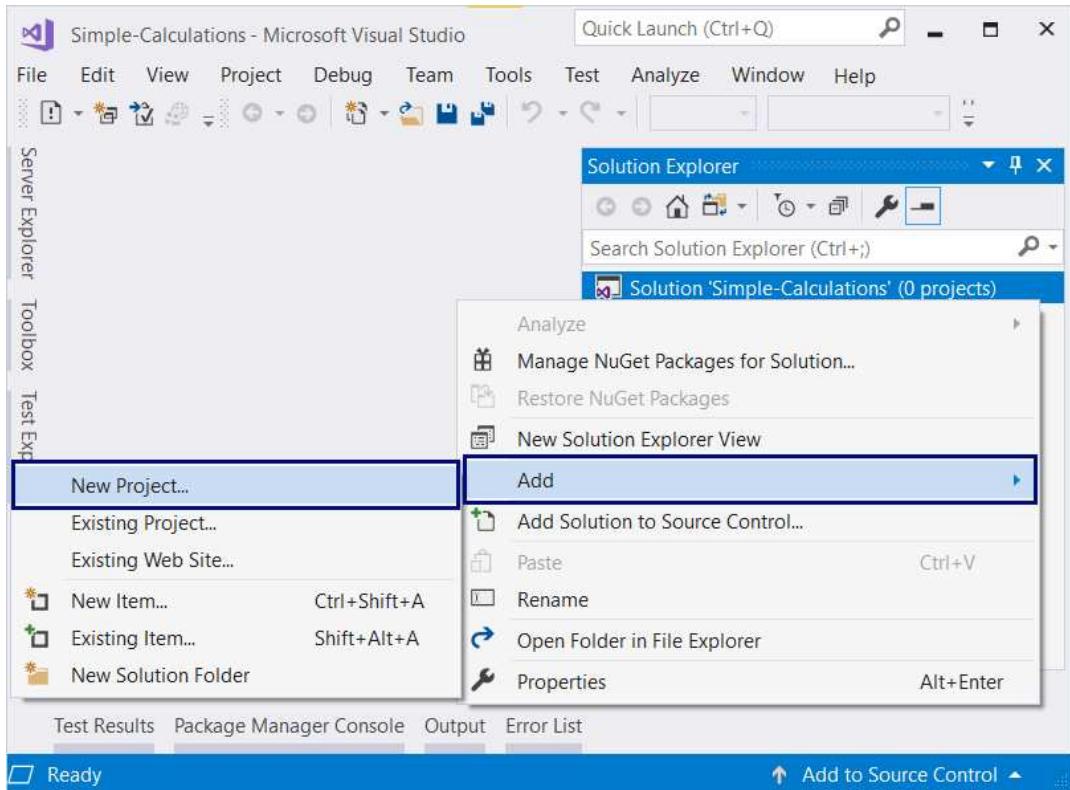
The purpose of this blank solution is to add a **project per problem** from the exercises.

Problem: Calculating Square Area

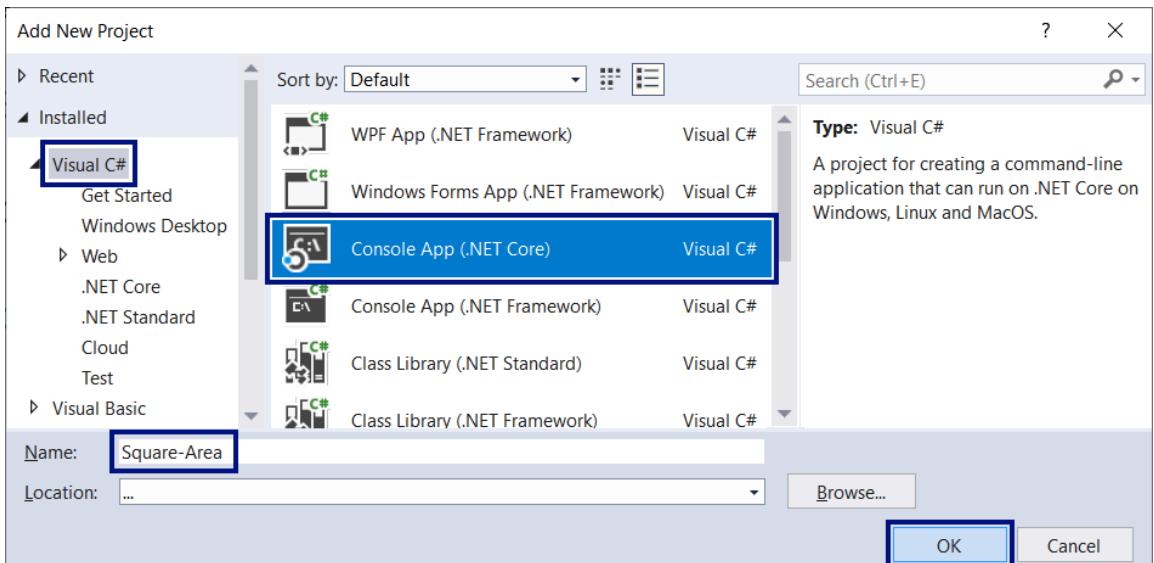
The first exercise from this topic is the following: write a console program that **inputs an integer a** and **calculates the area** of a square with side **a**. The task is trivial and easy: **input a number** from the console, **multiply it by itself** and **print the obtained result** on the console.

Hints and Guidelines

We create a **new project** in the existing Visual Studio solution. In the **Solution Explorer** right-click on Solution 'Simple-Calculations'. Choose [Add] -> [New Project...]:



A dialogue window is going to be opened for choosing the **project type** for creation. We choose **C# console application** with name "Square-Area":



We already have a solution with one console application in it. What remains is to write the **code for solving this problem**. For this purpose, we go to the main method's body `Main(string[] args)` and write the following code:

```

class SquareArea
{
    static void Main()
    {
        Console.Write("a = ");
        var a = int.Parse(Console.ReadLine());
        var area = a * a;
        Console.Write("Square = ");
        Console.WriteLine(area);
    }
}

```

The code inputs an integer using the code `a = int.Parse(Console.ReadLine())`, afterwards it calculates `area = a * a` and finally prints the value of the variable `area`. We start the program with [Ctrl+F5] and test it with different input values (see the screenshot).

```

C:\WINDOWS\system32\cmd.exe
a = 5
Square = 25
Press any key to continue . . .

```

Testing in the Judge System

Test your solution here: <https://judge.softuni.org/Contests/Practice/Index/504#0>. You have to get 100 points (completely correct solution):

01. Square Area

```

1 using System;
2
3 class SquareArea
4 {
5     static void Main()
6     {
7         Console.Write("a = ");
8         var a = int.Parse(Console.ReadLine());
9         var area = a * a;
10        Console.Write("Square = ");
11        Console.WriteLine(area);
12    }
13 }

```

Allowed working time: 0.100 sec. C# code **Submit**

Allowed memory: 16.00 MB
Size limit: 16.00 KB
Checker: Numbers Checker

Submissions		
Points	Time and memory used	Submission date
4/4 ✓✓✓✓ 100 / 100	Memory: 7.36 MB Time: 0.015 s	11:33:31 05.04.2019
		Details

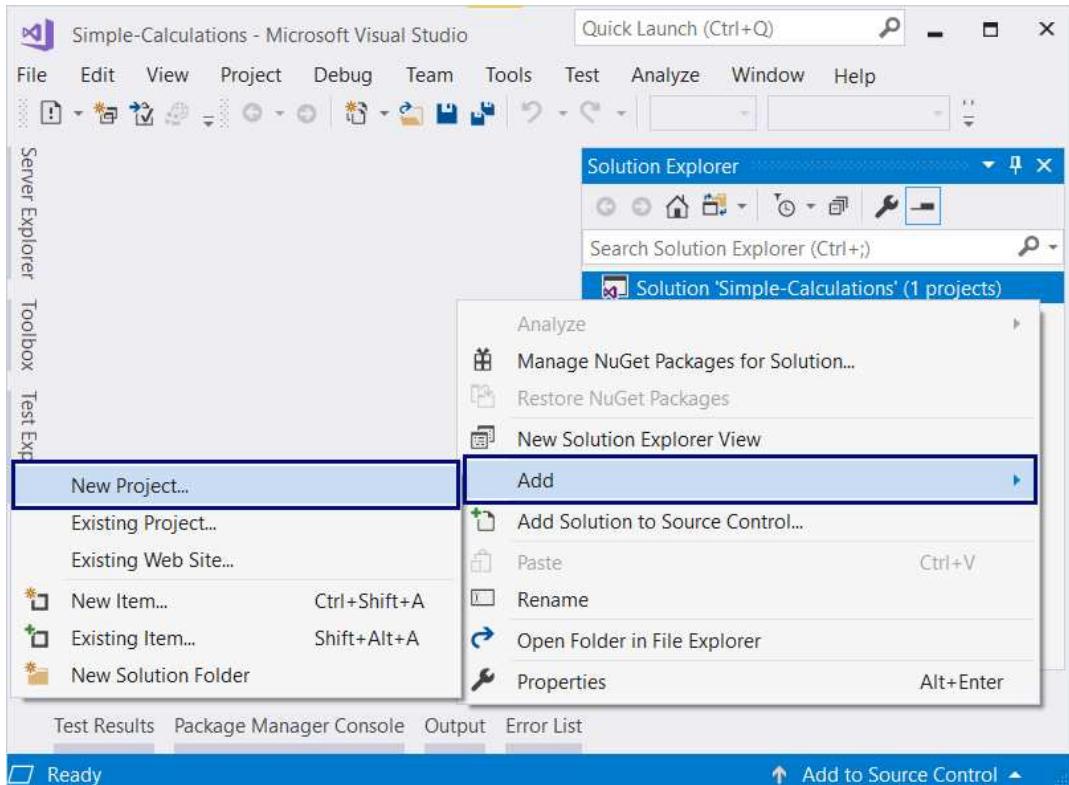
◀ ▶ 1

Problem: Inches to Centimeters

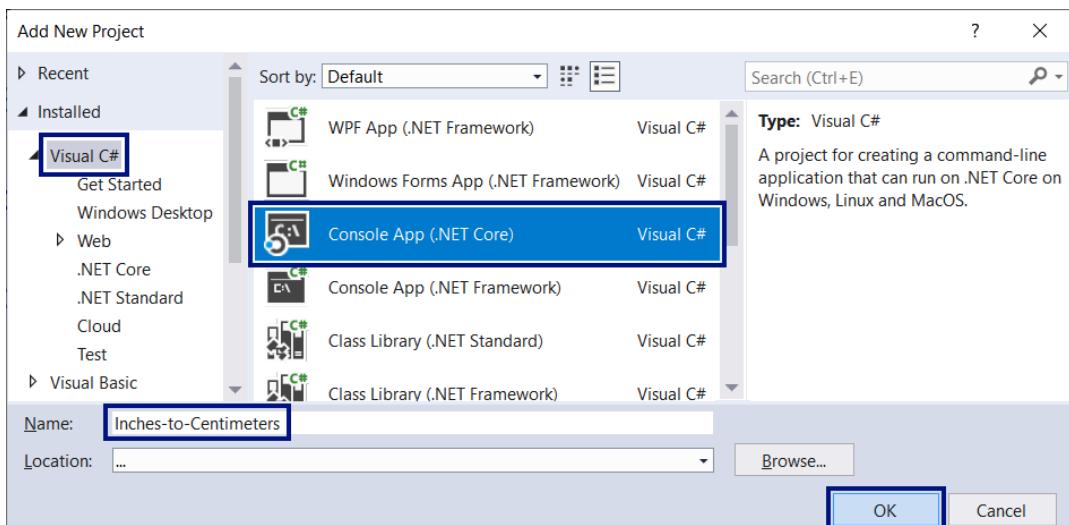
Write a program that reads a number from the console (not necessarily an integer) and converts the number from inches to centimeters. For the purpose it multiplies the inches by 2.54 (because one inch = 2.54 centimeters).

Hints and Guidelines

First, we create a new C# console project in the solution "Simple-Calculations". We right-click the solution in the Solution Explorer and we choose [Add] -> [New Project...]:



Select [Visual C#] -> [Windows] -> [Console Application] and name it "Inches-to-Centimeters":

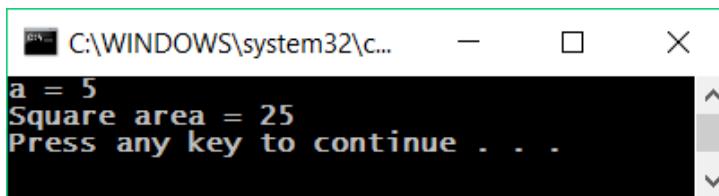


Writing Program Code and Starting the Program

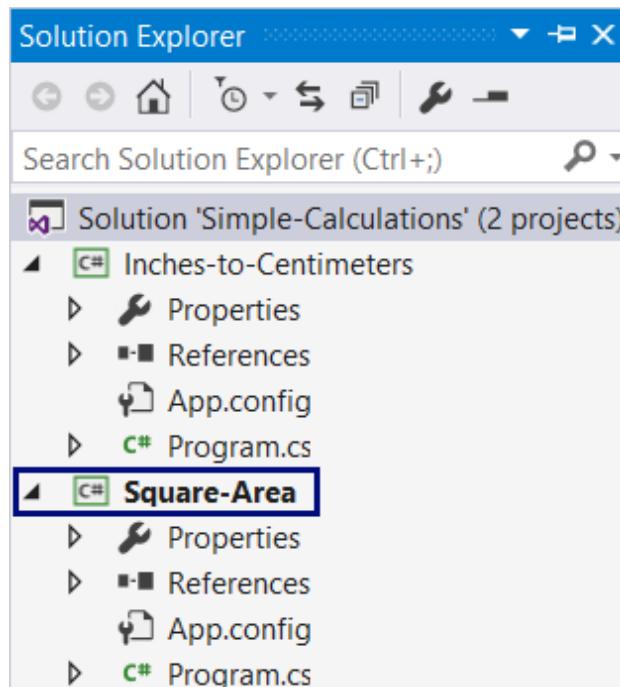
Next, we have to write the **program code**:

```
static void Main()
{
    Console.WriteLine("Inches = ");
    var inches = double.Parse(Console.ReadLine());
    var centimeters = inches * 2.54;
    Console.WriteLine("Centimeters = ");
    Console.WriteLine(centimeters);
}
```

Start the program with [Ctrl+F5]. The screenshot shows a sample execution.

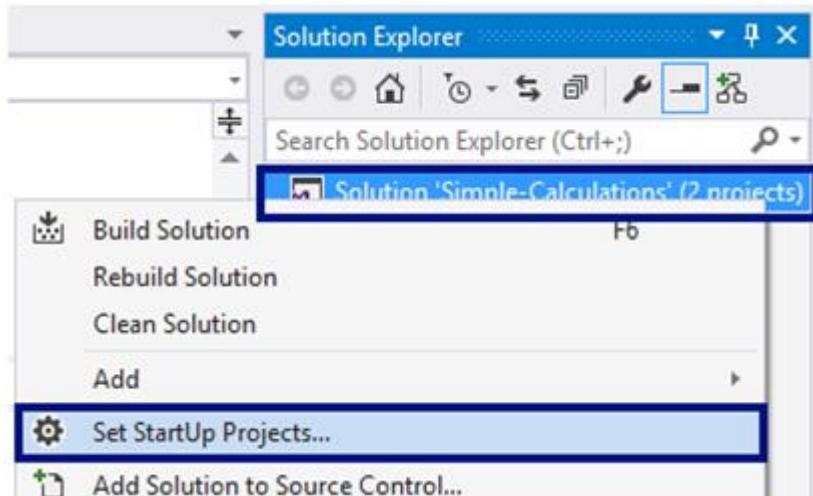


Surprise! What is happening? The program doesn't work correctly... Actually, isn't this the previous program? In Visual Studio **the current active project** in a solution is marked in semi-black color and it could be changed:

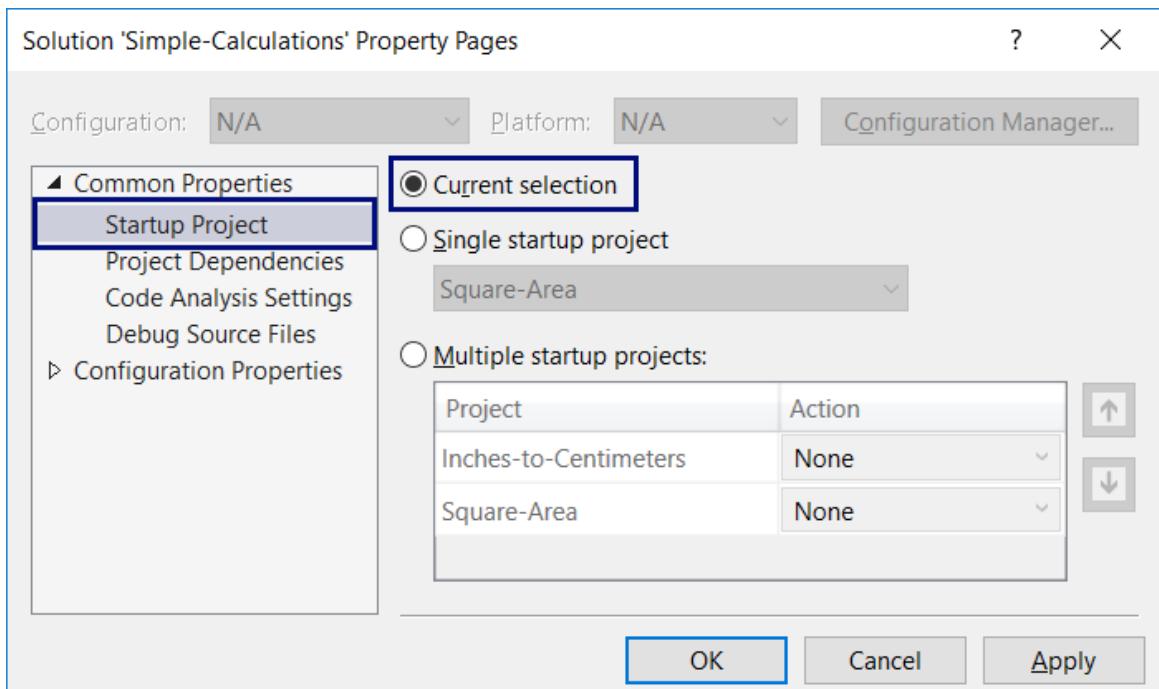


Setting Up a Startup Project

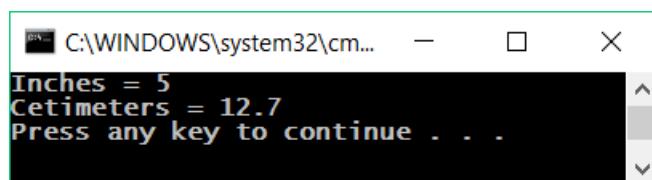
To switch the mode to **automatically go to current project**, we right-click the main solution and we choose **[Set StartUp Projects...]**:



A dialog window will open, and you will have to choose [Startup Project] -> [Current Selection]:



And again, we run the program, as usual with [Ctrl+F5]. This time it will start the current open program, which converts inches to centimeters. It looks like it works correctly:



Switching Between Programs

Now let's switch to the previous program (square area). This happens with a double click on the file **Program.cs** from the previous project "Square-Area" in the panel [Solution Explorer] in Visual Studio:

```

using System;

class SquareArea
{
    static void Main()
    {
        Console.WriteLine("a = ");
        var a = int.Parse(Console.ReadLine());
        var area = a * a;
        Console.WriteLine("Square = ");
        Console.WriteLine(area);
    }
}

```

We press again [Ctrl+F5]. This time the other project should start:

```

C:\WINDOWS\system3...
a = 3
Square area = 9
Press any key to continue . . .

```

We switch back to the "Inches-to-Centimeters" project and start it with [Ctrl+F5]:

```

C:\WINDOWS\system32\cm...
Inches = 10
Centimeters = 25.4
Press any key to continue . . .

```

Switching between projects is very easy, isn't it? Just choose the file with the source code of the program, double click it and when it starts, the program from the current file is being executed.

Testing a Program Locally

Let's test with floating point numbers, for example with 2.5:

```

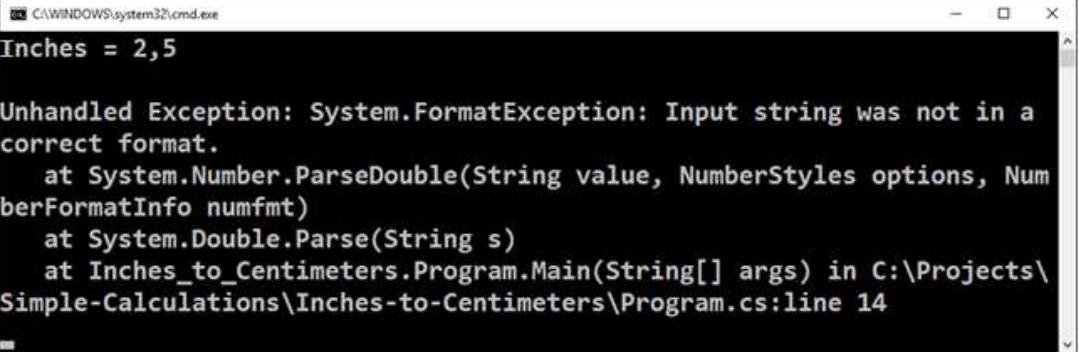
C:\WINDOWS\system32\c...
Inches = 2.5
Centimeters = 6.35
Press any key to continue . . .

```



Depending on the regional settings of the operation system, it is possible instead of using a **decimal point** (US settings), to use a **decimal comma** (BG settings).

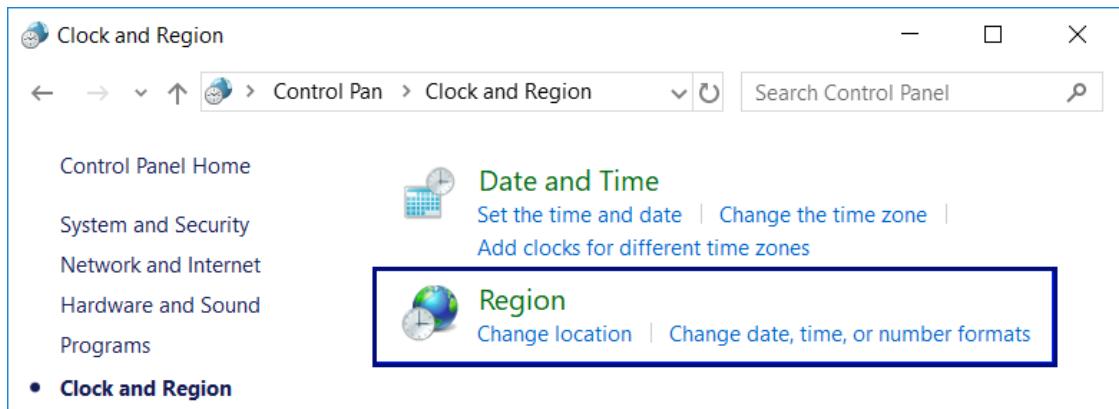
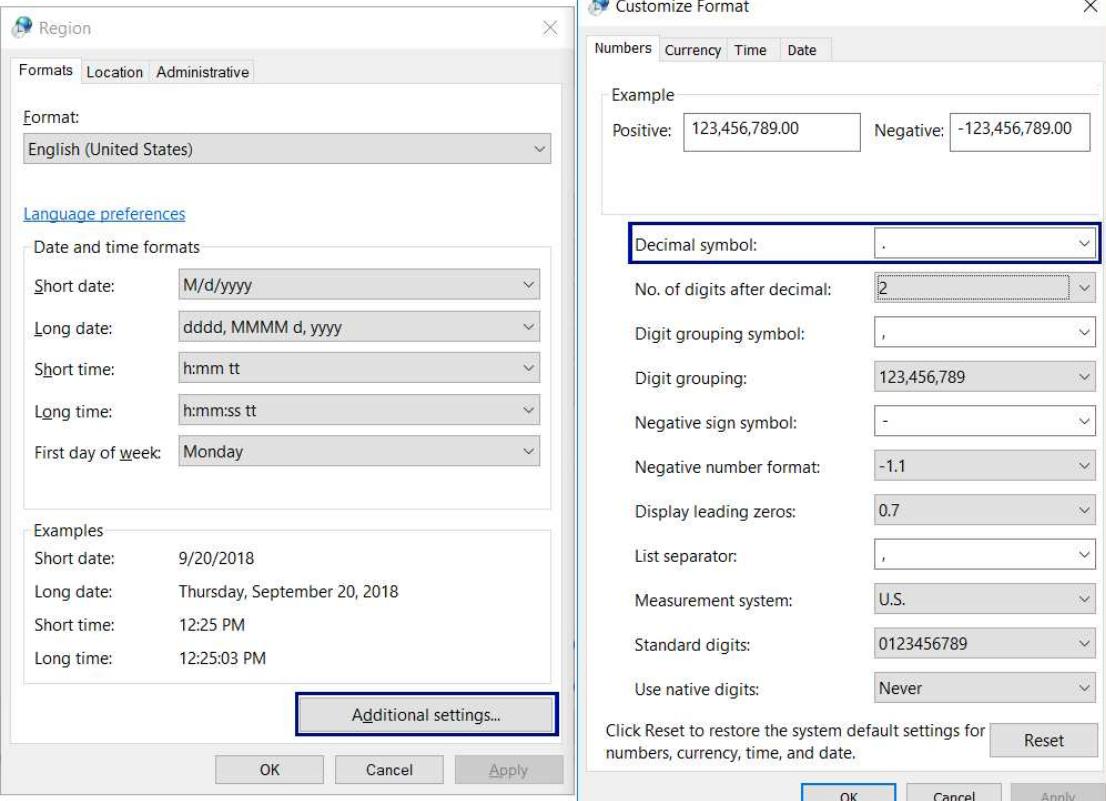
If the program expects a **decimal point** and instead a number with a **decimal comma** you enter the opposite (to enter a decimal point, when a decimal comma is expected), an error will be produced:



```
Inches = 2,5

Unhandled Exception: System.FormatException: Input string was not in a
correct format.
   at System.Number.ParseDouble(String value, NumberStyles options, Num
berFormatInfo numfmt)
   at System.Double.Parse(String s)
   at Inches_to_Centimeters.Program.Main(String[] args) in C:\Projects\Simple-Calculations\Inches-to-Centimeters\Program.cs:line 14
```

It is recommended to change the settings of your computer to use a decimal point:

The screenshot shows two windows side-by-side. On the left is the 'Region' control panel window, which includes sections for 'Formats', 'Language preferences', and 'Examples'. The 'Formats' tab is selected. On the right is the 'Customize Format' dialog box, which has tabs for 'Numbers', 'Currency', 'Time', and 'Date'. The 'Numbers' tab is selected, showing various format options like 'Decimal symbol', 'No. of digits after decimal', and 'Digit grouping'.

Testing in the Judge System

Test your solution here: <https://judge.softuni.org/Contests/Practice/Index/504#1>. The solution should be accepted as a completely **correct**:

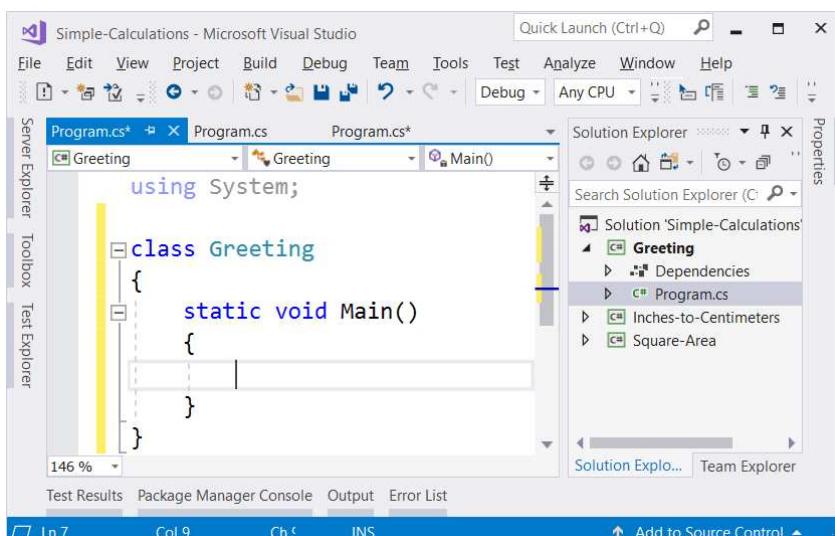
Submissions		
Points	Time and memory used	Submission date
100 / 100	Memory: 7.36 MB Time: 0.015 s	11:33:31 05.04.2019
1		Details

Problem: Greeting by Name

Write a program that reads a person's name and prints **Hello, <name>!**, where **<name>** is the name entered earlier.

Hints and Guidelines

First, we create a new C# console project with name "Greeting" in the solution "Simple-Calculations":



Next, we have to write the code of the program. If you have any difficulties, you can use the code from the example below:

```
static void Main()
{
    var name = Console.ReadLine();
    Console.WriteLine("Hello, {0}!", name);
}
```

Run the program with [Ctrl+F5] and test if it works:

```
C:\WINDOWS\system32\cmd.exe
Svetlin Nakov
Hello, Svetlin Nakov!
Press any key to continue . . .
```

Testing in the Judge System

Test your solution here: <https://judge.softuni.org/Contests/Practice/Index/504#2>.

Problem: Concatenating Text and Numbers

Write a C# program, that reads a first name, last name, age and city from the console and prints a message of the following kind: You are <firstName> <lastName>, a <age>-years old person from <town>.

Hints and Guidelines

We add to the existing Visual Studio solution one more console C# project with name “Concatenate-Data”. We **write the code** which reads the input from the console:

```
var firstName = Console.ReadLine();
var lastName = Console.ReadLine();
var age = int.Parse(Console.ReadLine());
var town = Console.ReadLine();
```

The code that prints the message described in the problem requirements should be finished.



In the picture above the code is **blurred on purpose**, in order for you to think of a way to finish it yourself.

Next, the solution should be tested locally using [Ctrl+F5] and by entering a sample input data.

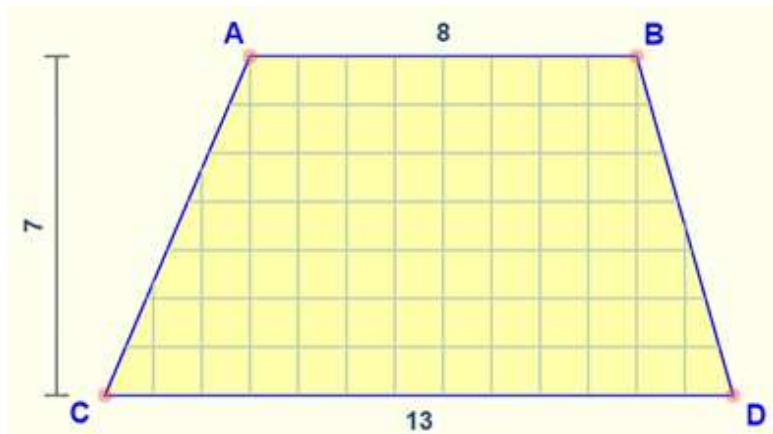
Testing in the Judge System

Test your solution here: <https://judge.softuni.org/Contests/Practice/Index/504#3>.

Problem: Trapezoid Area

Write a program that reads three numbers from the console **b1**, **b2** and **h** and calculates the area of a trapezoid with bases **b1** and **b2** and height **h**. The formula for trapezoid area is $(b_1 + b_2) * h / 2$.

The figure below shows a trapezoid with bases 8 and 13 and height 7. It has an area $(8 + 13) * 7 / 2 = 73.5$.



Hints and Guidelines

Again, we have to add to the existing Visual Studio solution another **console C# project** with name "Trapezoid-Area" and write the code that reads the input from the console, calculates the trapezoid area and prints it:

```
var b1 = double.Parse(Console.ReadLine());
Console.WriteLine("Trapezoid area = " + area);
```

The code in the picture is purposely blurred, in order for you to give it a thought and finish it yourself.

Test your solution locally using [Ctrl+F5] and enter an exemplary data.

Testing in the Judge System

Test your solution here: <https://judge.softuni.org/Contests/Practice/Index/504#4>.

Problem: Circle Area and Perimeter

Write a program that reads from the console a **number r** and calculates and prints **the area and perimeter of a circle with radius r**.

Input and Output

Input	Output
3	Area = 28.2743338823081 Perimeter = 18.8495559215388

Input	Output
4.5	Area = 63.6172512351933 Perimeter = 28.2743338823081

Video: Circle Perimeter and Area

Watch the video lesson about calculating circle perimeter and area: <https://youtu.be/7W6teq9IVGU>.

Hints and Guidelines

For the calculations you may use the following formulas:

- `Area = Math.PI * r * r.`
- `Perimeter = 2 * Math.PI * r.`

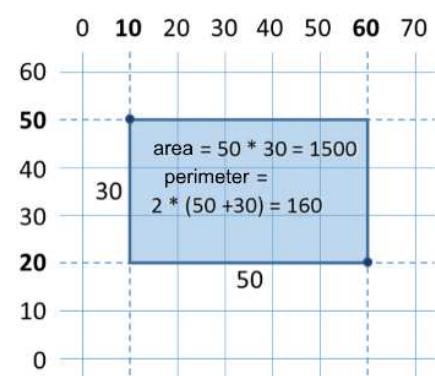
Testing in the Judge System

Test your solution in the SoftUni automated judge system:

<https://judge.softuni.org/Contests/Practice/Index/504#5>.

Problem: Rectangle Area

A rectangle is defined by the **coordinates** of both of its opposite corners (x_1, y_1) – (x_2, y_2). Calculate its **area and perimeter**. The **input** is read from the console. The numbers x_1, y_1, x_2 and y_2 are given one per line. The **output** is printed on the console and it has to contain two lines, each with one number – the area and the perimeter.



Sample Input and Output

Input	Output	Input	Output	Input	Output
60		30		600.25	350449.6875
20	1500	40	2000	500.75	2402
10	160	70	180	100.50	
50		-10		-200.5	

Video: Rectangle Area

Watch the video lesson about calculating rectangle area: https://youtu.be/lHb_Tz-EVT4.

Testing in the Judge System

Test your solution here: <https://judge.softuni.org/Contests/Practice/Index/504#6>.

Problem: Triangle Area

Write a program that reads from the console a side and height of a triangle and calculates its area. Use the formula for triangle area: $\text{area} = a * h / 2$. Round the result to 2 digits after the decimal point using `Math.Round(area, 2)`.

Sample Input and Output

Input	Output	Input	Output
20 30	Triangle area = 300	15 35	Triangle area = 262.5
7.75 8.45	Triangle area = 32.74	1.23456 4.56789	Triangle area = 2.82

Testing in the Judge System

Test your solution here: <https://judge.softuni.org/Contests/Practice/Index/504#7>.

Problem: Converter – from °C Degrees to °F Degrees

Write a program that reads degrees on Celsius scale ($^{\circ}\text{C}$) and converts them to degrees on Fahrenheit scale ($^{\circ}\text{F}$). Look on the Internet for a proper [formula](#) to do the calculations. Round the result to 2 digits after the decimal point.

Here are a few examples:

Sample Input and Output

Input	Output	Input	Output	Input	Output	Input	Output
25	77	0	32	-5.5	22.1	32.3	90.14

Testing in the Judge System

Test your solution here: <https://judge.softuni.org/Contests/Practice/Index/504#8>.

Problem: Converter – from Radians to Degrees

Write a program, that reads an angle in [radians](#) (rad) and converts it to [degrees](#) (deg). Look for a proper formula on the Internet. The number π in C# programs is available through [Math.PI](#). Round the result to the nearest integer using the [Math.Round\(...\)](#) method.

Sample Input and Output

Input	Output	Input	Output	Input	Output	Input	Output
3.1416	180	6.2832	360	0.7854	45	0.5236	30

Testing in the Judge System

Test your solution here: <https://judge.softuni.org/Contests/Practice/Index/504#9>.

Problem: Converter – USD to BGN

Write a program for conversion of US dollars (USD) into Bulgarian levs (BGN). Round the result to 2 digits after the decimal point, like it is shown at the examples below. Use a fixed rate between a dollar (USD) and levs (BGN): 1 USD = 1.79549 BGN.

Sample Input and Output

Input	Output	Input	Output	Input	Output
20	35.91 BGN	100	179.55 BGN	12.5	22.44 BGN

Testing in the Judge System

Test your solution here: <https://judge.softuni.org/Contests/Practice/Index/504#10>.

Problem: * Currency Converter

Write a program for conversion of money from one currency into another. It has to support the following currencies: BGN, USD, EUR, GBP. Use the following fixed currency rates:

Rate	USD	EUR	GBP
1 BGN	1.79549	1.95583	2.53405

The input is sum for conversion, input currency and output currency. The output is one number – the converted value of the above currency rates, rounded 2 digits after the decimal point.

Sample Input and Output

Input	Output	Input	Output	Input	Output	Input	Output
20 USD BGN	35.91 BGN	12.35 EUR GBP	9.53 GBP	100 BGN EUR	51.13 EUR	150.35 USD EUR	138.02 EUR

Testing in the Judge System

Test your solution here: <https://judge.softuni.org/Contests/Practice/Index/504#11>.

Problem: ** Date Calculations – 1000 Days on the Earth

Write a program that enters a birth date in format **dd-MM-yyyy** and calculates the date on which 1000 days are turned since this birth date and prints it in the same format.

Sample Input and Output

Input	Output
25-02-1995	20-11-1997
07-11-2003	02-08-2006

Input	Output
14-06-1980	10-03-1983
01-01-2012	26-09-2014

Input	Output
30-12-2002	24-09-2005
09-11-2003	05-08-2006

Hints and Guidelines

- Look for information about the data type **DateTime** in C# and in particular look at the methods **ParseExact(str, format)**, **AddDays(count)** and **ToString(format)**. With their help you can solve the problem without the need to calculate days, months and leap years.
- **Don't print** anything additional on the console except for the wanted date!

Testing in the Judge System

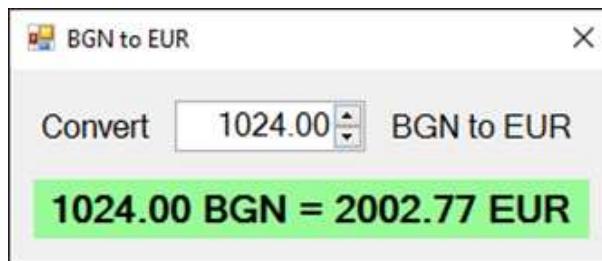
Test your solution here: <https://judge.softuni.org/Contests/Practice/Index/504#12>.

Lab: GUI Applications with Numerical Expressions

To exercise working with variables and calculations with operators and numerical expressions, we will make something interesting: we will develop a **desktop application** with graphical user interface. In it, we will use calculations with floating point numbers.

Graphical Application: Converter from BGN to EUR

We need to create a **graphical application** (GUI application) that calculates the value in **Euro (EUR)** of monetary amount given in **Bulgarian levs (BGN)**. By changing the amount in BGN, the amount in EUR has to be recalculated automatically (we use a fixed rate BGN / EUR: 1.95583).



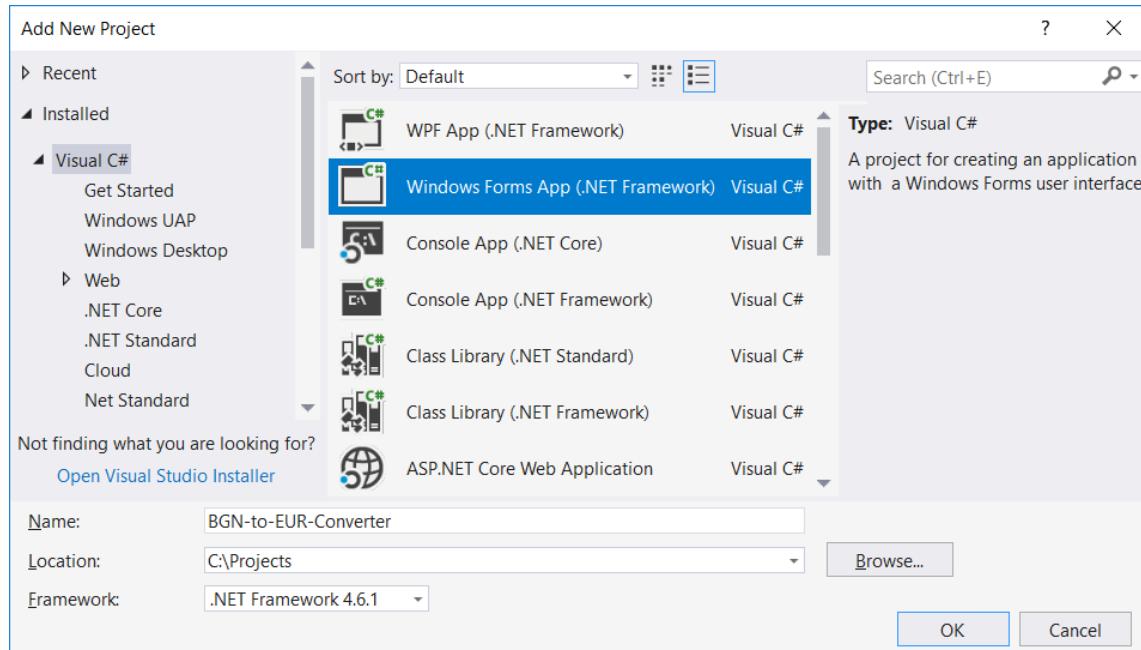
This exercise goes beyond the material learned in this book and its purpose is not to teach you how to program GUI applications, but to strengthen your interest in software development. Let's get to work.

Video: GUI Converter from BGN to EUR

Watch the following video lesson to learn how to build the Windows Forms based GUI application to convert BGN to EUR: <https://youtu.be/xWbDjzLsu9U>.

Creating a New C# Project

We add to the existing Visual Studio solution one more project. This time we create a **Windows Forms** application with C# named "**BGN-to-EUR-Converter**":

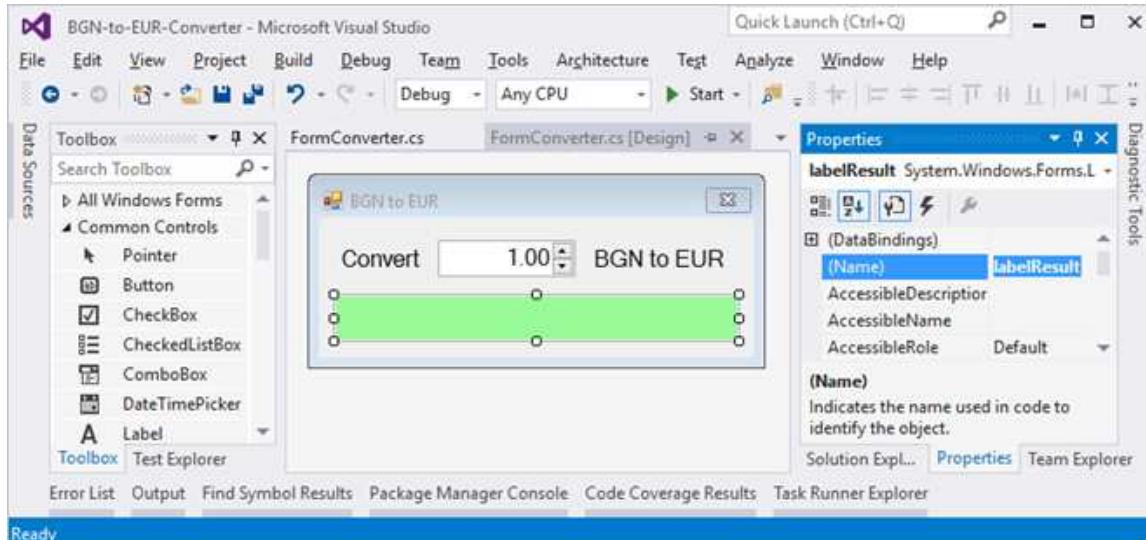


Adding UI Controls

We arrange the following UI (user interface) controls in the format:

- **NumericUpDown** with name **numericUpDownAmount** – it will enter the amount for conversion
- **Label** with name **labelResult** – it will show the result after conversion
- Two more **Label** components, serving only for static representation of a text

The graphical editor for user interface might look similar to this:

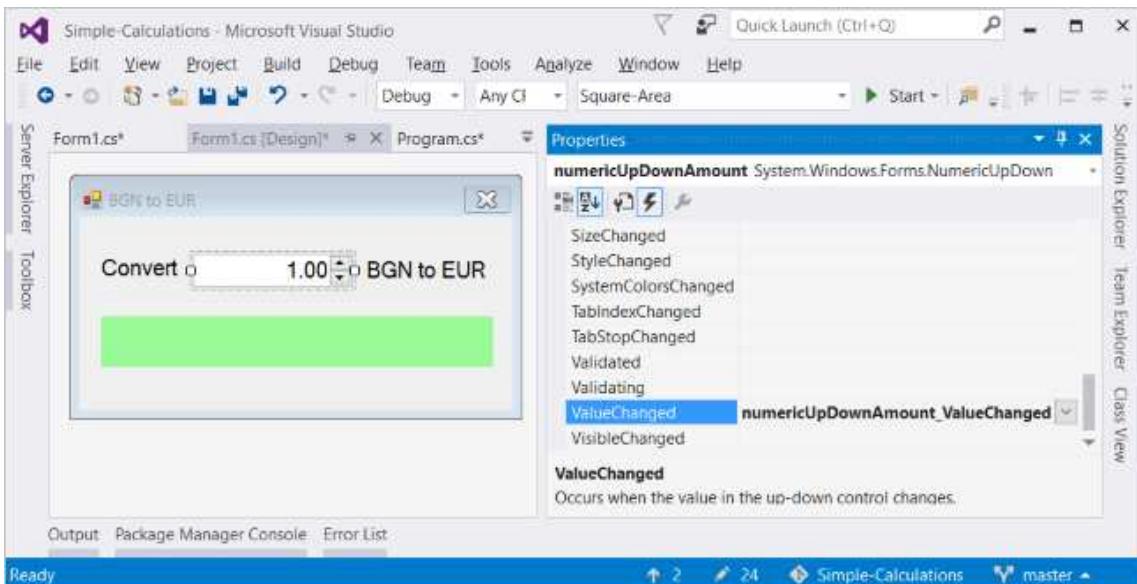


We do the following settings of the format and the separate controls:

Setting	Picture
FormConverter: Text = "BGN to EUR", FontSize = 12, MaximizeBox = False, MinimizeBox = False, FormBorderStyle = FixedSingle	
numericUpDownAmount: Value = 1, Minimum = 0, Maximum = 10000000, TextAlign = Right, DecimalPlaces = 2	
labelResult: AutoSize = False, BackColor = PaleGreen, TextAlign = MiddleCenter, FontSize = 14, FontBold = True	

Events and Event Handlers

We define the following **event handlers** in the controls:



After this we catch the following events:

- **FormConverter.Load** (by double-clicking with the mouse)
- **numericUpDownAmount.ValueChanged** (by double-clicking on **NumericUpDown** control)
- **numericUpDownAmount.KeyUp** (we choose **Events** from the dashboard **Properties** and double-click on **KeyUp**)

The event **Form.Load** is executed when the program is started, before the app window is shown. The event **NumericUpDown.ValueChanged** is executed when a change in the value inside the field for entering a number occurs. The event **NumericUpDown.KeyUp** is executed after pressing a key in the field that enters a number. On the occurrence of each of these events, we will recalculate the result. To **catch an event**, we use the events icon (Events) in the [Properties] window in Visual Studio:



Writing the Program Code

We will use the following **C# code** for handling events:

```
private void FormConverter_Load(object sender, EventArgs e)
{
    ConvertCurrency();
}

private void numericUpDownAmount_ValueChanged(object sender, EventArgs e)
{
    ConvertCurrency();
}

private void numericUpDownAmount_KeyUp(object sender, KeyEventArgs e)
{
    ConvertCurrency();
}
```

All of the caught events call the method **ConvertCurrency()**, which converts the given sum from BGN to EUR and shows the result in the green box. We have to write the **code** (program logic) for the conversion:

```
private void ConvertCurrency()
{
    var amountBGN = this.numericUpDownAmount.Value;
    var amountEUR = amountBGN * 1.95583m;
    this.labelResult.Text = amountBGN + " BGN = " +
        Math.Round(amountEUR, 2) + " EUR";
}
```

Testing the Application

Finally, we start the project with [Ctrl+F5] and test if it works correctly.

If you have any problems with the example project above, you can ask in the SoftUni official **discussion forum** (<http://forum.softuni.org>) or in the SoftUni official **Facebook page** (<https://fb.com/softuni.org>).

Graphical Application: * Catch the Button!

Create a fun graphical application “**catch the button**”: a form consisting of one button. Upon moving the mouse cursor onto the button, it moves to a random position. This way it creates the impression

that "the button runs from the mouse and it is hard to catch". When the button gets "caught", a congratulations message is shown.



Hints and Guidelines

Write an event handler `Button.MouseEnter` and move the button to a random position. Use the random numbers generator `Random`. The position of the button is set using the `Location` property. To make sure the new position of the button is within the form's borders, you can make calculations based on the size of the form, which is available from the `ClientSize` property.

You may use the following [sample code](#):

```
private void buttonCatchMe_MouseEnter(object sender, EventArgs e)
{
    Random rand = new Random();
    var maxWidth = this.ClientSize.Width - buttonCatchMe.ClientSize.Width;
    var maxHeight = this.ClientSize.Height - buttonCatchMe.ClientSize.Height;
    this.buttonCatchMe.Location = new Point(
        rand.Next(maxWidth), rand.Next(maxHeight));
}
```

Be active, be curious, experiment, play, learn, enjoy!

Useful Web Sites for C# Developers

C# developers use Internet resources constantly in their daily job. They search in Internet, read developer's news, sites and blogs and discuss technical questions with their colleagues.

In this video lesson we shall give you a few popular Web sites for C# developers:
<https://youtu.be/DdNo8iofjHw>.

The Web sites mentioned in the video are very helpful for the beginners in C# programming and are highly recommended:

- <https://www.dotnetperls.com>
- <https://stackoverflow.com>
- <https://docs.microsoft.com/dotnet/csharp>

High-quality education profession and job for software engineers

- ✓ Free start for **absolute beginners**
- ✓ Choose among the **most in-demand** professions in the software industry
- ✓ Assistance for **career start**
- ✓ **Top trainers**, mentors and active learning **community**



SoftUni provides practical, modern and **innovative education** for programming, IT, design, marketing and digital skills for thousands of young people. The "**Software University**" program builds true **software engineering professionals**.

Join the end-to-end **software engineering learning program** at SoftUni to master the most in-demand programming concepts, languages, software technologies and skills through a modern teaching methodology, **interactive learning platform** and tons of practical exercises and projects. Our **curriculum** is developed together with the **IT companies** to closely match the industry demands.

SoftUni works directly with the **companies** from the software industry to **find a job** for its students and to make them successful software engineers.

The path of the student at SoftUni



Apply now

softuni.org/apply

Chapter 2.2. Simple Calculations – Exam Problems

In the previous chapter, we explained how to work with the system console – how to **read numbers** from it and how to **print the output**. We went through the main arithmetical operations and briefly mentioned data types. Now, we are going to practice what we have learned by solving a few **more complex exam problems**.

Simple Calculations – Quick Review

Before jumping into the practical problems, let's first make a **quick review** of what we learned in the previous chapter.

Reading Numbers from the Console

Before going to the tasks, we are going to revise the most important aspects of what we have studied in the previous chapter. We will start by reading numbers from the console.

Reading an Integer

We need to create a variable to store the integer (for example, **num**) and use a standard command for reading input from the console **Console.ReadLine()**, combined with the function **int.Parse(...)** which converts string to an integer:

```
var num = int.Parse(Console.ReadLine());
```

Reading a Floating-Point Number

We read a floating-point number, the same way we read an integer one, but this time we use the function **double.Parse(...)**:

```
var num = double.Parse(Console.ReadLine());
```

Printing Text Using Placeholders

Placeholder is an expression which is replaced with a particular value while printing an output. The methods **Console.WriteLine(...)** supports printing a string based on a placeholder, where the first argument is the formatted string, followed by the number of arguments, equal to the number of placeholders.

```
Console.WriteLine("You are {0} {1}, a {2}-years old person from {3}.",  
    firstName, lastName, age, town);
```

Arithmetic Operators

Let's revise the main arithmetic operators for simple calculations.

Operator +

```
var result = 3 + 5; // the result is 8
```

Operator -

```
var result = 3 - 5; // the result is -2
```

Operator *

```
var result = 3 * 5; // the result is 15
```

Operator /

```
var result = 7 / 3;      // the result is 2 (integer division)
var result2 = 5 / 2.0;   // the result is 2.5 (floating-point division)
```

String Concatenation

By using the operator **+** between string variables (or between a string and a number), **concatenation** is being performed (combining strings).

```
var firstName = "Ivan";
var lastName = "Ivanov";
var age = 19;
var str = firstName + " " + lastName + " is " + age + " years old";
// Ivan Ivanov is 19 years old
```

Exam Problems

Now, after having revised how to make simple calculations and how to read and print numbers from the console, let's go to the tasks. We will solve a few **problems** from a **SoftUni entrance exam**.

Problem: Training Lab

A **training lab** has a rectangular size **l** x **w** meters, without columns on the inside. The hall is divided into two parts: left and right, with a hallway approximately in the middle. In both parts, there are **rows with desks**. In the back of the hall, there is a big **entrance door**. In the front, there is a **podium** for the lecturer. A single **working place** takes up **70 x 120 cm** (a table with size 70 x 40 cm + space for a chair with size 70 x 80 cm). The **hallway** width is at least **100 cm**. It is calculated that due to the **entrance door** (which has 160 cm opening), **exactly one working space is lost**, and due to the **podium** (which has size of 160 x 120 cm), exactly **two working spaces** are lost. Write a program that reads the size of the training lab as input parameters and calculates the **number of working places in it** (look at the figure).

Input Data

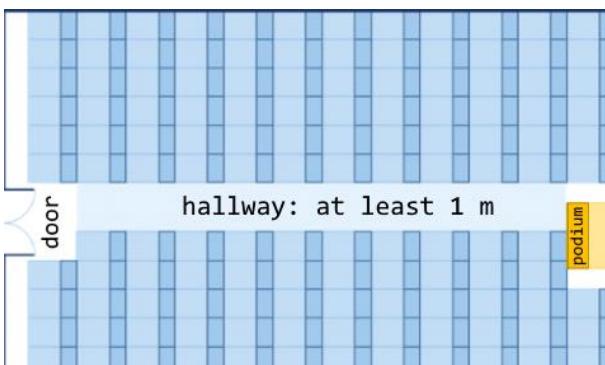
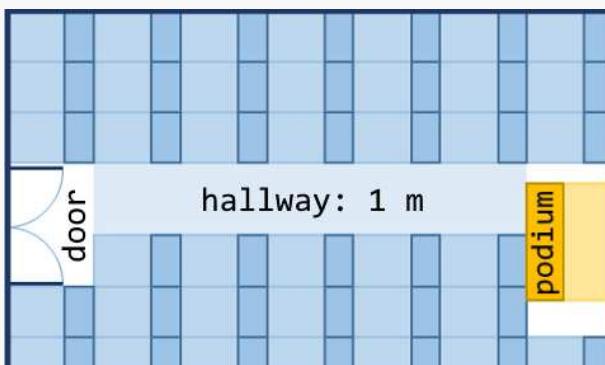
Two numbers are read from the console, one per line: **l** (length in meters) and **w** (width in meters).

Constraints: **3 ≤ w ≤ l ≤ 100**.

Output Data

Print an integer: the **number of working places** in the training lab.

Sample Input and Output

Input	Output	Figure
15 8.9	129	
8.4 5.2	39	

Clarification of the Examples

In the **first example**, the hall length is 1500 cm. **12 rows** can be situated in it ($12 * 120 \text{ cm} = 1440 + 60 \text{ cm difference}$). The hall width is 890 cm. 100 cm of them are for the hallway in the middle. The rest 790 cm can be situated by 11 desks per row ($11 * 70 \text{ cm} = 770 \text{ cm} + 20 \text{ cm difference}$). **Number of places = $12 * 11 - 3 = 132 - 3 = 129$** (we have 12 rows with 11 working places = 132 minus 3 places for podium and entrance door).

In the **second example**, the hall length is 840 cm. **7 rows** can be situated in it ($7 * 120 \text{ cm} = 840$, no difference). The hall width is 520 cm. 100 cm from them are for the hallway in the middle. The rest 420 cm can be situated by 6 desks per row ($6 * 70 \text{ cm} = 420 \text{ cm}$, no difference). **Number of places = $7 * 6 - 3 = 42 - 3 = 39$** (we have 7 rows with 6 working places = 42 minus 3 places for podium and entrance door).

Hints and Guidelines

Try to solve the problem on your own first. If you do not succeed, go through the hints.

Idea for Solution

As with any programming task, **it is important to build an idea for its solution**, before having started to write code. Let's carefully go through the problem requirements. We have to write a program that calculates the number of working places in a training lab, where the number depends on the hall length and height. We notice that the provided input will be **in meters** and the information about how much space the working places and hallway take, will be **in centimeters**. To do the calculations, we will use the same measuring units, no matter whether we choose to convert length and height into centimeters or the other data in meters. The first option is used for the presented solution.

Next, we have to calculate **how many columns** and **how many rows** with desks will fit. We can calculate the columns by **subtracting the width by the necessary space for the hallway (100 cm)** and **divide the difference by 70 cm** (the length of a working place). We find the rows by **dividing the length by 120 cm**. Both operations can result in **a real number** with whole and fractional part, but we have to **store only the whole part in a variable**. In the end, we multiply the number of rows by the number of columns and divide it by 3 (the lost places for entrance door and podium). This is how we calculate the needed value.

Choosing Data Types

From the example, we see that a real number with whole and fractional part can be given as an input, therefore, it is not appropriate to choose data type **int**. This is why we use **double**. Choosing data type for the next variables depends on the method we choose to solve the problem. As with any programming task, this one has **more than one way to be solved**. Two methods will be shown here.

Solution – Variant I

It is time to go to the solution. We can divide it into three smaller tasks:

- Reading input from the console.
- Doing the calculations.
- Printing the output on the console.

The first thing we have to do is read the input from the console. With **Console.ReadLine()** we read the values from the console and with the function **double.Parse(...)** string is converted into **double**.

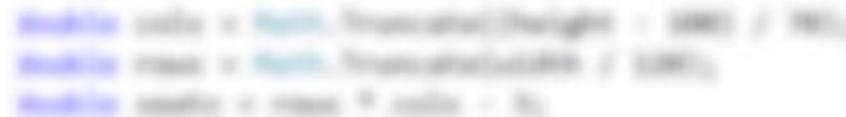
```
double length = double.Parse(Console.ReadLine());  
double width = double.Parse(Console.ReadLine());
```

Let's move to the calculations. The special part here is that after having divided the numbers, we have to store only the whole part of the result in a variable.



Search in Google! Whenever we have an idea how to solve a particular problem, but we do not know how to write it in C# or we are dealing with one that many other people have had before us, the easiest way to solve it is by looking for information on the Internet.

In this case, we can try with the following search: "[c# get whole number part of double](#)". One possible way is to use the method **Math.Truncate(...)** as it works with **double** data types. For the number of rows and columns we create variables of the same type.



Solution – Variant II

Second variant: As we already know, the operator for division **/** operates differently on integers and decimals. **When dividing integer with integer, the result is also an integer**. Therefore, we can search how to convert the real numbers that we have as values for the height and the width, into integers and then divide them.

In this case, there could be **data loss** after having removed the fractional part, so it is necessary that it is converted **expressly** (explicit typecasting). We use the operator for converting data (**type**) by replacing the word **type** with the needed **data type** and place it **before** the variable.

```
int cols = ((int)width - 100) / 70;
int rows = (int)length / 120;
int seats = rows * cols - 3;
```

With `Console.WriteLine(...)` we print the result on the console.

Testing in the Judge System

Test your solution here: <https://judge.softuni.org/Contests/Practice/Index/505#0>.

Problem: Vegetable Market

A gardener is selling his harvest on the vegetables market. He is selling **vegetables** for N coins per kilogram and **fruits** for M coins per kilogram. Write a program that calculates the earnings of the harvest in Euro (EUR). Assume the EUR / coin rate is fixed: 1 Euro == 1.94 coins.

Input Data

Four numbers are read from the console, one per line:

- First line: **vegetable price** per kilogram – a floating-point number.
- Second line: **fruit price** per kilogram – a floating-point number.
- Third line: total **kilograms of vegetables** – an integer.
- Fourth line: total **kilograms of fruits** – an integer.

Constraints: all numbers will be within the range from 0.00 to 1000.00.

Output Data

Print on the console one floating-point number: the earnings of all fruits and vegetables in Euro.

Sample Input and Output

Input	Output	Explanation	Input	Output
0.194 19.4 10 10	101	Vegetables cost: 0.194 coins * 10 kg = 1.94 coins Fruits cost: 19.4 coins * 10 kg = 194 coins Total: 195.94 coins = 101 euro (= 101 * 1.94)	1.5 2.5 10 10	20.6185567010309

Hints and Guidelines

First, we will give a few **ideas** and particular hints for solving the problem, followed by the essential part of the **code**.

Idea for Solution

Let's first go through the problem requirements. In this case, we have to calculate the **total income** from the harvest. It equals **the sum of the earnings from the fruits and vegetables** which we can calculate by multiplying **the price per kilogram by the quantity**. The input is given in coins and the

output should be in EUR. It is assumed that 1 Euro equals 1.94 coins, therefore, in order to get the wanted output value, **we have to divide the sum by 1.94**.

Choosing Data Types

After we have a clear idea of how to solve the task, we can continue with choosing appropriate data types. Let's go through the **input**: we have **two integers** for total kilograms of vegetables and fruits, therefore, the variables we declare to store their values will be of **int** type. The prices of the fruits and vegetables are said to be floating-point numbers and therefore, the variables will be **double**.

We can also declare two variables to store the income from the fruits and vegetables separately. As we are multiplying a variable of **int** type (total kilograms) with one of **double** type (price), the result should also be of **double** type. Let's clarify that: generally, **operators work with arguments of the same type**. Therefore, in order to multiply values of different data types, we have to convert them into the same type. When there are types of different scopes in one expression, the one with the highest scope is the one the other types are converted to, in this case, **double**. As there isn't danger of data loss, **the conversion is implicit** and is automatically done by the compiler.

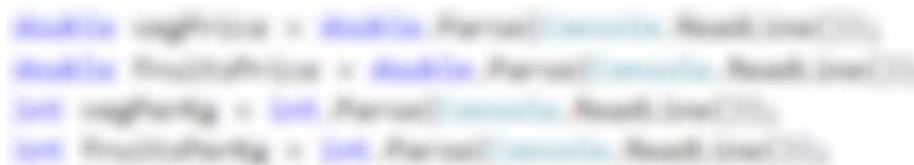
The **output** should also be a **floating-point number** and therefore, the result will be stored in a variable of **double** type.

Solution

It is time to get to the solution. We can divide it into three smaller tasks:

- **Reading input from the console.**
- **Doing the calculations.**
- **Printing the output on the console.**

In order to read the input, we declare variables, which we have to name carefully, so that they can give us a hint about the values they store. With **Console.ReadLine(...)**, we read values from the console and with the functions **int.Parse(...)** and **double.Parse(...)**, we convert the particular string value into **int** and **double**.



We do the necessary calculations:

```
double vegTotal = vegPrice * vegPerKg;
double fruitTotal = fruitsPrice * fruitsPerKg;
```

The task does not specify special output format. Therefore, we just have to calculate the requested value and print it on the console. As in mathematics and so in programming, division has a priority over addition. However, in this task, first we have to **calculate the sum** of the two input values and then **divide by 1.94**. In order to give priority to addition, we can use brackets in the expression. With **Console.WriteLine(...)** we print the output on the console.

```
Console.WriteLine((fruitTotal + vegTotal) / 1.94);
```

Testing in the Judge System

Test your solution here: <https://judge.softuni.org/Contests/Practice/Index/505#1>.

Problem: Change Tiles

The tiles on the ground in front of an apartment building need changing. The ground has a **square shape with side of N meters**. The tiles are "**W**" meters wide and "**L**" meters long. There is one bench on the ground with width of "**M**" meters and length of "**O**" meters. The tiles under it do not need to be replaced. Each tile is replaced for **0.2 minutes**. Write a program that **reads the size of the ground, the tiles and the bench from the console**, and calculates how many tiles are needed to cover the ground and what is the total time for replacing the tiles.

Example: ground with size 20 m has area of 400 m². A bench that is 1 m wide and 2 m long, has area of 2 m². One tile is 5 m wide, 4 m long and has area of 20 m². The space that needs to be covered is $400 - 2 = 398$ m². Therefore, $398 / 20 = 19.90$ tiles are necessary. The time needed is $19.90 * 0.2 = 3.98$ minutes.

Input Data

The input data comes as **5 numbers**, which are read from the console:

- **N** – length of a side of the ground within the range of [1 ... 100].
- **W** – width per tile within the range of [0.1 ... 10.00].
- **L** – length per tile within the range of [0.1 ... 10.00].
- **M** – width of the bench within the range of [0 ... 10].
- **O** – length of the bench within the range of [0 ... 10].

Output Data

Print on the console **two numbers**: number of tiles needed for the repair and the **total time for changing them**, each on a new line.

Sample Input and Output

Input	Output	Comments
20		Total area = $20 * 20 = 400$.
5		Area of the bench = $1 * 2 = 2$.
4	19.9	Area for covering = $400 - 2 = 398$.
1	3.98	Area of tiles = $5 * 4 = 20$.
2		Needed tiles = $398 / 20 = 19.9$.
		Needed time = $19.9 * 0.2 = 3.98$.

Input	Output
40	
0.8	3302.08333333333
0.6	660.416666666667
3	
5	

Hints and Guidelines

Let's make a draft to clarify the task requirements. It can look like at the figure.

Idea for Solution

It is required to calculate **the number of tiles** that have to be changed, as well as **the total time for replacing them**. In order to find the **number of tiles**, we have to calculate the **area that needs to be covered** and **divide it by the area per tile**. The ground is square, therefore, we find the total area by multiplying its side by its value $N * N$. After that, we calculate **the area that the bench takes up** by multiplying its two sides as well $M * O$. After subtracting the area of the bench from the area of the whole ground, we obtain the area that needs to be repaired.

We calculate the area of a single tile by **multiplying its two sides with one another $W * L$** . As we already stated, now we have to **divide the area for covering by the area of a single tile**. This way, we

find the number of necessary tiles which we multiply by **0.2** (the time needed for changing a tile). Now, we have the wanted output.

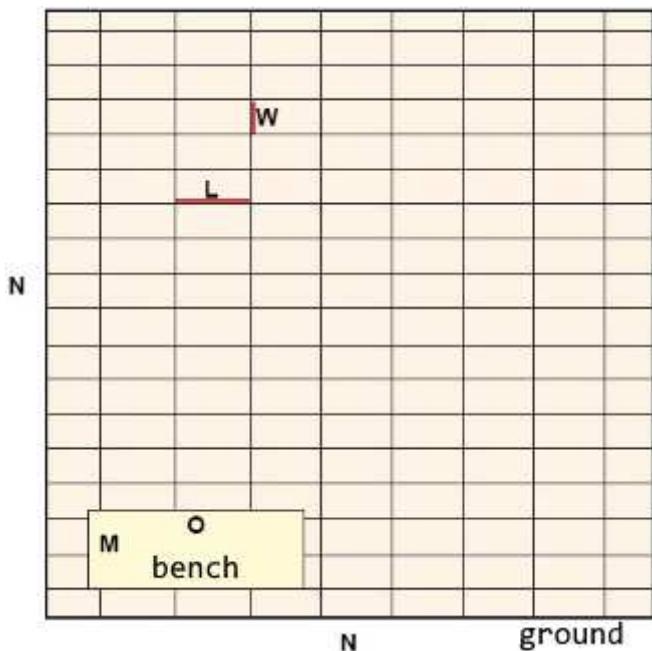
Choosing Data Types

The length of the side of the ground, the width and the length of the bench, will be given as **integers**, therefore, in order to store their values, we can declare **variables of int type**. We will be given floating-point numbers for the width and the length of the tiles and this is why we will use **double**. The output will be a floating-point number as well, so the variables will be of **double** type as well.

Reading the Input Data

As in the previous tasks, we can divide the solution into three smaller tasks:

- Reading the input from the console.
- Doing the calculations.
- Printing the output on the console.

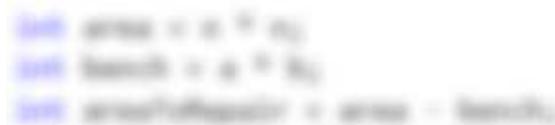


The first thing we have to do is go through the **input** of the task. It is important to pay attention to the sequence they are given in. With **Console.ReadLine()** we read values from the console and with **int.Parse(...)** and **double.Parse(...)**, we convert the string values into **int** or **double**.

```
// Ground length
int n = int.Parse(Console.ReadLine());
// Tile width
double w = double.Parse(Console.ReadLine());
// Tile length
double h = double.Parse(Console.ReadLine());
// Bench width
int a = int.Parse(Console.ReadLine());
// Bench length
int b = int.Parse(Console.ReadLine());
```

Performing the Calculations

After we have initialized the variables and have stored the corresponding values in them, we move to the **calculations**. As the values of the variables **n**, **a** and **b** are stored in variables of type **int**, we can also declare **variables of the same type** for the results.



The variables **w** and **h** are of type **double**, therefore, for the **area of a single tile**, we create a variable of the same type. Finally, **we calculate the values that we have to print on the console**. The number of necessary **tiles** is obtained by dividing the area that needs to be covered by the area of a tile. When

dividing the two numbers, one of which is a **floating-point number**, the result will also be a **floating-point number**. Therefore, in order for the calculations to be correct, we store the result in a variable of type **double**. The task does not specify special formatting or rounding of the output, so we just print the values with `Console.WriteLine(...)`.

```
double tiles = areaToRepair / (w * h);

double time = tiles * 0.2;

Console.WriteLine(tiles);
Console.WriteLine(time);
```

Testing in the Judge System

Test your solution here: <https://judge.softuni.org/Contests/Practice/Index/505#2>.

Problem: Money

Peter lives in Bulgaria and is keen about money exchange, trading, cryptocurrencies and financial markets. Some time ago, **Peter bought Bitcoins** and paid for them in **Bulgarian levs** (BGN). Now, he is going on vacation in Europe and **he needs Euro** (EUR). Apart from the Bitcoins, he has **Chinese yuan** (CNY) as well. Peter wants to **exchange his money for Euro** for the tour. Write a program that calculates **how much Euro** he can buy, depending on the following **exchange rates**:

- 1 Bitcoin (BTC) = 1168 BGN
- 1 Chinese yuan (CNY) = 0.15 dollars (USD)
- 1 Dollar (USD) = 1.76 BGN
- 1 Euro (EUR) = 1.95 BGN

The exchange office has **commission fee within 0% to 5%** from the final sum in Euro.

Input Data

Three numbers are read from the console:

- On the first line – **number of Bitcoins**. Integer within the range of [0 ... 20].
- On the second line – **number of Chinese yuan**. Floating-point number within the range of [0.00 ... 50 000.00].
- On the third line – **commission fee**. Floating-point number within the range of [0.00 ... 5.00].

Output Data

Print one number on the console – **the result of the exchange of currencies**. Rounding is not necessary.

Sample Input and Output

Input	Output	Explanation
1 5 5	569.668717948718	1 Bitcoin (BTC) = 1168 BGN 5 Chinese yuan (CNY) = 0.75 dollars 0.75 dollars (USD) = 1.32 BGN $1168 + 1.32 = 1169.32 \text{ BGN} = 599.651282051282 \text{ Euro}$ (EUR)

Input	Output	Explanation
		<p>Commission fee: 5% of 599.651282051282 = 29.9825641025641</p> <p>Result: 599.651282051282 - 29.9825641025641 = 569.668717948718 Euro</p>

Input	Output	Input	Output
20 5678 2.4	12442.24	7 50200.12 3	10659.47

Hints and Guidelines

Let's first think of the way we can solve the task again, before having started to write code.

Idea for Solution

We see that the **number of bitcoins** and the **number of Chinese yuans** will be given in the input. The **output** should be in **Euro**. The exchange rates that we have to work with are specified in the task. We notice that we can only exchange the sum in BGN to EUR, therefore, we **first have to calculate the whole sum that Peter has in BGN, and then calculate the output**.

As we have information for the exchange rate of bitcoins to BGN, we can directly exchange them. On the other hand, in order to get the value of **Chinese yuans in BGN**, first we have to **exchange them in dollars**, and then **the dollars to BGN**. Finally, we will **sum the two values** and calculate how much Euro that is.

Only the final step is left: **calculating the commission fee** and subtracting the new sum from the total one. We will obtain **an integer** for the commission fee, which will be a particular **percent from the total sum**. Let's divide it by 100, so as to calculate its **percentage value** and then multiply it by the sum in Euro. We will divide the result from the same sum and print the final sum on the console.

Choosing Data Types

Bitcoins are given as **an integer**, therefore, we can declare a **variable of int type** for their value. For **Chinese yuan and commission fee** we obtain **a floating-point number**, therefore, we are going to use **double**. As **double** is the data type with bigger scope, and the **output** should also be **a floating-point number**, we will use it for the other variables we create as well.

Solution

After we have built an idea on how to solve the task and we have chosen the data structures that we are going to use, it is time to get to **writing the code**. As in the previous tasks, we can divide the solution into three smaller tasks:

- **Reading input from the console.**
- **Doing the calculations.**
- **Printing the output** on the console.

We declare the **variables** that we are going to use and again we have to choose **meaningful names**, which are going to give us hints about the values they store. We initialize their values: with **Console.ReadLine(...)**, we read the input numbers from the console and convert the string entered by the user to **int** or **double**.

```
decimal bitcoins = int.Parse(Console.ReadLine());
decimal yuans = decimal.Parse(Console.ReadLine());
decimal commission = decimal.Parse(Console.ReadLine()) / 100.00m;
```

We do the necessary calculations:

```
decimal bitcoinsToBGN = bitcoins * 1168;
decimal yuansToDollars = yuans * 0.15m;
decimal dollarsToBGN = yuansToDollars * 1.76m;
```

```
decimal bitcoins = 100;
decimal yuans = 1000;

decimal bitcoinsToBGN = bitcoins * 1168;
decimal yuansToDollars = yuans * 0.15m;
decimal dollarsToBGN = yuansToDollars * 1.76m;
```

Finally, we calculate the commission fee value and subtract it from the sum in Euro. Let's pay attention to the way we could write this: `Euro -= commission fee * Euro` is the short way to write `Euro = Euro - (commission fee * Euro)`. In this case, we use a **combined assignment operator** (`-=`) that **subtracts the value of the operand to the right from the one to the left**. The operator for multiplication (`*`) has a **higher priority** than the combined operator, this is why, the expression `commission fee * Euro` is performed first and then its value is divided.

The task does not specify special string formatting or rounding the result, therefore, we just have to calculate the output and print it on the console.

```
euro -= commission * euro;
Console.WriteLine(euro);
```

Let's pay attention to something that applies to all other problems of this type: written like that, the solution of the task is pretty detailed. As the task itself is not too complex, in theory, we could write one big expression, where right after having taken the input, we calculate the output. For example, such expression would look like this:

```
double euro = (bitcoins * 1168 + yuans * 0.15 * 1.76) / 1.95
    - ((bitcoins * 1168 + yuans * 0.15 * 1.76) / 1.95 * commission);
```

This code would print a correct result, but it is **hard to read**. It won't be easy to find out how it works and whether it contains any mistakes, as well as finding such and correcting them. **Instead of one complex expression, it is better to write a few simpler ones** and store their values in variables with appropriate names. This way, the code is cleaner and easily maintainable.

Testing in the Judge System

Test your solution here: <https://judge.softuni.org/Contests/Practice/Index/505#3>.

Problem: Daily Earnings

Ivan is a programmer in an American company, and he **works** at home approximately **N** days per month by earning **approximately M** dollars per day. At the end of the year, Ivan **gets a bonus**, which **equals 2.5 of his monthly salaries**. In addition, 25% of his annual salary goes for taxes.

Write a program that calculates **what is the amount of Ivan's net average earnings in EUR per day**, as he spends them in Europe. It is assumed that **one year has exactly 365 days**. The **exchange rate of US dollar (USD) to Euro (EUR)** will be **read from the console**.

Input Data

Three numbers are read from the console.

- On the first line – **workdays per month**. An integer within the range of [5 ... 30].
- On the second line – **daily earnings**. A floating-point number within the range of [10.00 ... 2000.00].
- On the third line – **exchange rate of USD to EUR**: 1 dollar = X euro. A floating-point number within the range of [0.05 ... 4.99].

Output Data

Print **one number** on the console – the daily earnings in EUR. The result should be rounded up to the second digit after the decimal point.

Sample Input and Output

Input	Output	Input	Output	Input	Output
21		15		22	
75.00	41.30	105	80.24	199.99	196.63
0.88		1.71		1.50	

Explanation for the first example:

- One monthly salary = $21 * 75.00 = 1575$ dollars.
- Annual income = $1575 * 12$ months + $1575 * 2.5$ bonus = 22837.5 dollars.
- Taxes = 25% of 22837.5 = 5709.375 dollars.
- Net annual income in USD = $22837.5 - 5709.375 = 17128.125$ dollars.
- Net annual income in EUR = 17128.125 dollars * 0.88 = 15072.75 EUR.
- Average earnings per day = $15072.75 / 365 \approx 41.30$ EUR.

Hints and Guidelines

Firstly, we have to analyze the problem and think of a way to solve it. Then, we will choose data types and, finally, we will write the code.

Idea for Solution

Let's first calculate **how much the monthly salary** of Ivan is. We do that by **multiplying the working days per month by his daily earnings**. Firstly, we multiply the number **by 12**, so as to calculate his salary for 12 months, and then, we multiply it **by 2.5** in order to calculate the bonus. After having summed up the two values, we calculate his **annual income**. Then, we reduce the annual income **by 25% taxes**. Depending on the exchange rate, we **exchange the dollars (USD) to Euro (EUR)** and after that we **divide the result by 365** (days per year).

Choosing Data Types

The **working days** per month are given as **an integer**, therefore, we can declare a variable of **int type** to store their value. For both **the earned money** and the **exchange rate of USD to EUR**, we will obtain **a floating-point number**, therefore, we will use **double**. As **double** is the data type with **the higher scope**, and the output should also be **a floating-point number**, we use **double** for the other variables that we create as well.

Reading the Input Data

Again: after we have an idea on how to solve the problem and we have considered the data types that we are going to use, we can start **writing the program**. As in the previous tasks, we can divide the solution into three smaller tasks:

- Reading the input from the console.
- Doing the calculations.
- Printing the output on the console.

We declare the variables that we are going to use by trying to choose **meaningful names**. With `Console.ReadLine(...)` we read the input numbers from the console and we **convert** the input string to `int` or `double` with `int,double.Parse(...)`.



Doing the Calculations

We do the calculations:

```
double monthSalary = workdays * moneyPerDay;
double moneyPerYear = (monthSalary * 12) + (monthSalary * 2.5);
double taxes = 0.25 * moneyPerYear;
double netSalary = moneyPerYear - taxes;
double salaryInEUR = netSalary * currencyRate;
```

We could write an expression that calculates the annual income without brackets as well. As multiplication is an operation that has a higher priority over addition, it will be performed first. Despite that, **writing brackets is recommended** when using more operators, as this way the code is **easily readable** and chances of making a mistake are smaller.

Printing the Result

Finally, we have to print the result on the console. We notice that **the number has to be rounded up to the second digit after the decimal point**. In order to do that, we can use a **placeholder** – an item that will be replaced by a particular value when printing. In C#, a digit surrounded by curly brackets is used for a placeholder. As in programming counting starts from 0, the expression `{0}` means that it will be replaced by the first given argument. We can format an integer or a floating-point number by using `F` or `f`, which is followed by a whole positive number, specifying the number of digits after the decimal point:

```
double average = salaryInEUR / 365;
Console.WriteLine("{0:f2}", average);
```

Testing in the Judge System

Test your solution here: <https://judge.softuni.org/Contests/Practice/Index/505#4>.

High-quality education profession and job for software engineers

- ✓ Free start for **absolute beginners**
- ✓ Choose among the **most in-demand** professions in the software industry
- ✓ Assistance for **career start**
- ✓ **Top trainers**, mentors and active learning **community**



SoftUni provides practical, modern and **innovative education** for programming, IT, design, marketing and digital skills for thousands of young people. The "**Software University**" program builds true **software engineering professionals**.

Join the end-to-end **software engineering learning program** at SoftUni to master the most in-demand programming concepts, languages, software technologies and skills through a modern teaching methodology, **interactive learning platform** and tons of practical exercises and projects. Our **curriculum** is developed together with the **IT companies** to closely match the industry demands.

SoftUni works directly with the **companies** from the software industry to **find a job** for its students and to make them successful software engineers.

The path of the student at SoftUni



Apply now

softuni.org/apply

Chapter 3.1. Simple Conditions

In this chapter, we will discuss the **conditional statements** in the C# language, through which our program may have different effects, depending on a condition. We'll explain the syntax of conditional operators for checks (**if** and **if-else**) with appropriate examples and we will see in what range a variable (its **scope**) lives. Finally, we will go through **debugging** techniques to track the path that runs through our program during implementation.

Video: Chapter Overview

Watch the video lesson about what we will learn in this chapter: <https://youtu.be/sstA00rlWk0>.

Introduction to Simple Conditions by Examples

In programming we can **check conditions** and execute different blocks of code depending on the check. This is typically performed using the **if-else** constructs:

```
var size = decimal.Parse(Console.ReadLine());
if (size < 0)
    Console.WriteLine($"Negative size: {size}");
else if (size > 1000)
    Console.WriteLine($"Size too big: {size}");
else
{
    Console.WriteLine($"Size accepted: {size}");
    Console.WriteLine($"Area: {size * size}");
}
```

Run the above code example: <https://repl.it/@nakov/size-checker-if-else-csharp>.

When executed, the above code will **enter a decimal number** and will **check its value** several times. Depending on the above **conditions**, it will display different messages. Examples are shown below.

If we **enter -20 as input**, the output will be as follows:

```
Negative size: -20
```

If we **enter 150 as input**, the output will be as follows:

```
Size accepted: 150
Area: 22500
```

If we **enter 3200 as input**, the output will be as follows:

```
Size too big: 3200
```

Let's explain in greater detail how to use **simple if-else conditions** in C#.

Comparing Numbers

In programming, we can compare values using the following **operators**:

- Operator **<** (less than)
- Operator **>** (greater than)

- Operator `<=` (less than or equals)
- Operator `>=` (greater than or equals)
- Operator `==` (equals)
- Operator `!=` (different than)

When compared, the result is a Boolean value `true` or `false`, depending on whether the result of the comparison is true or false.

Video: Comparing Numbers

Watch the video lesson about comparing numbers: <https://youtu.be/KTdqDWg7Wf8>.

Examples for Comparing Numbers

```
var a = 5;
var b = 10;
Console.WriteLine(a < b);      // True
Console.WriteLine(a > 0);      // True
Console.WriteLine(a > 100);     // False
Console.WriteLine(a < a);       // False
Console.WriteLine(a <= 5);      // True
Console.WriteLine(b == 2 * a);   // True
```

Note that when printing the `true` and `false` values in C# language, they are printed with a capital letter, respectively `True` and `False`.

Comparison Operators

In C#, we can use the following **comparison operators**:

Operator	Notation	Applicable for
Greater than	<code>></code>	
Greater than or equals	<code>>=</code>	numbers, dates, other comparable objects
Less than	<code><</code>	
Less than or equals	<code><=</code>	
Equals	<code>==</code>	numbers, strings, dates
Not equal	<code>!=</code>	

The following **example** demonstrates how to use comparison operators in expressions:

```
var result = (5 <= 6);
Console.WriteLine(result); // True
```

Simple If Conditions

In programming we often **check particular conditions** and perform various actions depending on the result of the check. This is done by **if** condition, which has the following structure:

```
if (condition)
{
    // condition body;
}
```

Video: Simple If / If-Else Conditions

Watch the video lesson about the simple if-conditions: <https://youtu.be/MG4nOaVDVA>.

Example: Excellent Grade

We read the grade from the console and check if it's excellent (≥ 5.50).

```
var grade = double.Parse(Console.ReadLine());
if (grade >= 5.50)
{
    Console.WriteLine("Excellent!");
}
```

Test the code from the example locally. Try entering different grades, for example 4.75, 5.49, 5.50 and 6.00. For grades **less than 5.50**, the program will not give any output, however if the grade is **5.50 or greater**, the output would be "Excellent!".

Testing in the Judge System

Test your solution here: <https://judge.softuni.org/Contests/Practice/Index/506#0>.

If-Else Conditions

The **if** construction may also contain an **else** clause to give a specific action in case the Boolean expression (which is set at the beginning **if (bool expression)**) returns a negative result (**false**). Built this way, the **conditional statement** is called **if-else** and its behavior is as follows: if the result of the condition is **positive (true)** – we perform some actions, when it is **negative (false)** – others. The format of the construction is:

```
if (condition)
{
    // condition body;
}
else
{
    // else construction body;
}
```

Example: Excellent Grade or Not

Like the example above, we read the grade from the console and check if it's excellent, but this time we should **output the result in both cases**.

```
var grade = double.Parse(Console.ReadLine());
if (grade >= 5.50)
{
```

```

        Console.WriteLine("Excellent!");
    }
else
{
    Console.WriteLine("Not excellent.");
}

```

Testing in Judge System

Test your solution here: <https://judge.softuni.org/Contests/Practice/Index/506#1>.

About the Curly Braces {} After If / Else

When we have **only one command** in the body of the **if construction**, we can **skip the curly brackets**, indicating the conditional operator body. When we want to execute **block of code** (group of commands), curly brackets are **required**. In case we drop them, **only the first line** after the **if clause** will be executed.



It's a good practice to **always put curly braces**, because it makes our code more readable and cleaner.

Here is an example where dropping curly braces leads to confusion:

```

var color = "red";
if (color == "red")
    Console.WriteLine("tomato");
else if (color == "yellow")
    Console.WriteLine("banana");
Console.WriteLine("lemon");

```

Executing the above code will output the following console result:

```

C:\WINDOWS\system3...
tomato
lemon
Press any key to continue . . .

```

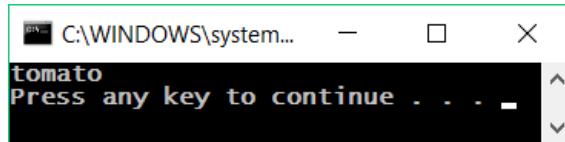
With curly braces:

```

var color = "red";
if (color == "red")
{
    Console.WriteLine("tomato");
}
else if (color == "yellow")
{
    Console.WriteLine("banana");
    Console.WriteLine("lemon");
}

```

The following output will be printed on the console:



If-Else Conditions – Examples

Now let's take a few examples (exercises) to learn how to use **if-else** conditional statements in practice.

Video: Examples of If-Else

Watch the video lesson about the "Even or Odd Number" and "The Larger Number" problems and their solutions: <https://youtu.be/qIuOkiObr-A>.

Example: Even or Odd Number

Write a program that checks whether an integer is **even** or **odd**.

Hint and Guidelines

We can solve the problem with one **if-else** statement and the operator **%**, which returns a **remainder** by dividing two numbers.

```
var num = int.Parse(Console.ReadLine());
if (num % 2 == 0)
{
    Console.WriteLine("even");
}
else
{
    Console.WriteLine("odd");
}
```

Testing in the Judge System

Test your solution here: <https://judge.softuni.org/Contests/Practice/Index/506#2>.

Example: The Larger Number

Write a program that reads from the console **two integers** and prints the **larger** of them. Print "Greater number: " + the bigger number.

The **input** comes as two numbers, each on a separate line. Sample input:

```
3
5
```

The **output** is a message like the shown below. Sample output:

```
Greater number: 5
```

Hint and Guidelines

Our first task is to **read** the two numbers from the console. Then, with a simple **if-else** statement, combined with the **operator for greater than (>)**, we do check. Part of the code is deliberately blurred, so you can test what you learned so far.

```
Console.WriteLine("Enter two integers:");
var num1 = int.Parse(Console.ReadLine());
var num2 = int.Parse(Console.ReadLine());
```

Testing in the Judge System

Test your solution here: <https://judge.softuni.org/Contests/Practice/Index/506#3>.

Variable Scope

Each variable has a range in which it exists, called **variable scope**. This range specifies where a variable can be used and how long is its **lifetime**. In the C# language, the scope in which a variable exists, starts from the line in which we **defined it** and ends with the first closing curly bracket } (of the method, the **if statement**, etc.).

Thus, it is important to know that **any variable defined inside the body of certain if statement will not be available outside of it**, unless we have defined it higher in the code.

Video: Variable Scope

Watch the video lesson about the variable scope: <https://youtu.be/J54TJBnY5vQ>.

Variable Scope – Example

In the example below, on the last line we are trying to print the variable **salary** that is defined in the **if statement**, we will get an **error** because we don't have access to it.

```
var myMoney = 500;
var payDayDate = 07;
var todayDate = 10;
if (todayDate >= payDayDate)
{
    var salary = 5000;
    myMoney = myMoney + salary;
}

Console.WriteLine(myMoney);
Console.WriteLine(salary); //Error!
```

The above code **will not compile**, because we are trying to access a variable **out of its scope**. The scope of the **salary** variable is limited inside the block after the **if**, starting from **{** and ending by **}**.

Sequence of If-Else Conditions

Sometimes we need to do a sequence of conditions before we decide what actions our program will execute. In such cases, we can apply the construction **if-else if ... -else in series**. For this purpose, we use the following “chained if-else” format:

```
if (condition)
{
    // condition body;
}
else if (second condition)
{
    // condition body;
}
else if (third condition)
{
    // condition body;
}
...
else
{
    // else construction body;
}
```

Video: Series of If-Else Checks

Watch the video lesson about the if-else checks: <https://youtu.be/PUvf7gtKSz4>.

Example: Digits in English

Print the digits in the range of 1 to 9 (digits are read from the console) in English. We can read the digit and then, through a **sequence of conditions** we print the relevant English word:

```
int num = int.Parse(Console.ReadLine());
if (num == 1)
{
    Console.WriteLine("one");
}
else if (num == 2)
{
    Console.WriteLine("two");
}
else if (...)
{
    ...
}
else if (num == 9)
{
    Console.WriteLine("nine");
}
else
{
    Console.WriteLine("number too big");
}
```

The program logic from the above example **sequentially compares** the input number from the console with the digits from 1 to 9, when **each following comparison is being performed only in case the previous comparison is not true**. Eventually, if none of the **if** statements are true, the last **else** clause is performed.

Testing in the Judge System

Test your solution here: <https://judge.softuni.org/Contests/Practice/Index/506#4>.

Debugging: Simple Operations with Debugger

So far, we wrote a lot of code, and there were some mistakes in it, right? Now we will show a tool that can help us find mistakes more easily: the **debugger**.

Video: Debugging Code in Visual Studio

Watch the video lesson about debugging code in Visual Studio: <https://youtu.be/2asLYeJ6Tel>.

What is "Debugging"?

Debugging is the process of "**attaching**" to the program execution, which allows us to **track step by step the process**. We can track **line by line** what happens in our program, what path it follows, what are the values of defined variables at each step of debugging, and many other things that allow us to detect errors (**bugs**).

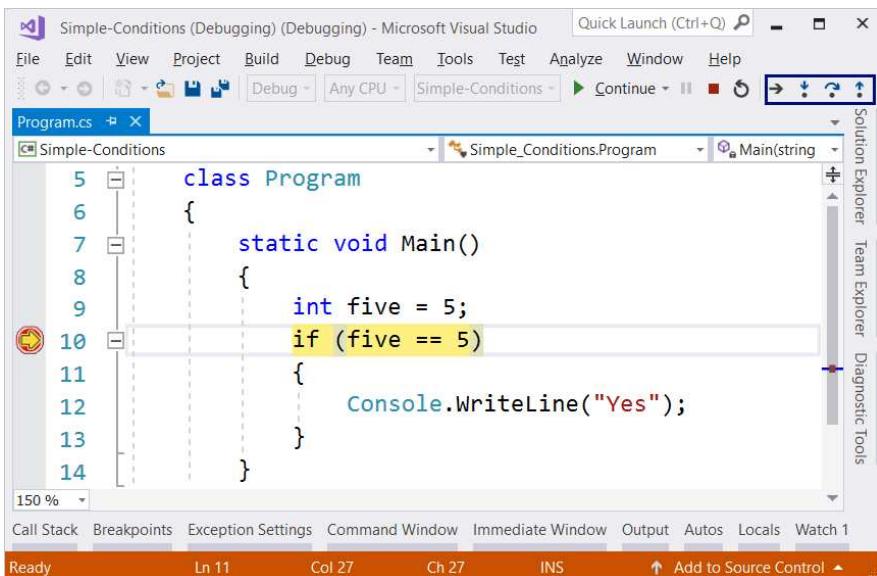
```

1  using System;
2
3  class Program
4  {
5      static void Main()
6      {
7          var number = int.Parse(Console.ReadLine());
8
9          number += number; ⏎ 4,319ms elapsed
10         Console.WriteLine(number);
11     }
12 }
```

Name	Value
System.Console.ReadLine returned	"10"
int.Parse returned	10
number	10

Debugging in Visual Studio

By pressing the [F10] button, we run the program in **debug mode**. We move to the **next line** again with [F10].



With [F9] we create the so-called **breakpoints**, that we can reach directly using [F5] when we start the program.

Exercises: Simple Conditions

Now let's **practice** the lessons learned in this chapter about of conditional statements **if** and **if-else**. We will solve a few **practical exercises**.

Video: Chapter Summary

Watch the following **video** to summarize what we learned in this chapter about the conditional statements in C#: <https://youtu.be/mdv28HD5ges>.

What We Learned in This Chapter?

Let's summarize what we learned in this chapter:

- Numbers can be **compared** by the `==`, `<`, `>`, `<=`, `>=` and `!=` operators:

```
Console.WriteLine(5 <= 10); // True
```

- Simple **if-conditions** check a condition and execute a code block if it is true:

```
if (a > 5)
{
    Console.WriteLine("The number `a` is bigger than 5");
}
```

- The **if-else construction** executes one of two blocks depending on whether a condition is **true** or **false**:

```
if (a > 5)
{
    Console.WriteLine("The number `a` is bigger than 5");
}
```

```

else
{
    Console.WriteLine("The number `a` is smaller or equal than 5");
}

```

- If-else constructions can be chained as if-else-if-else sequences:

```

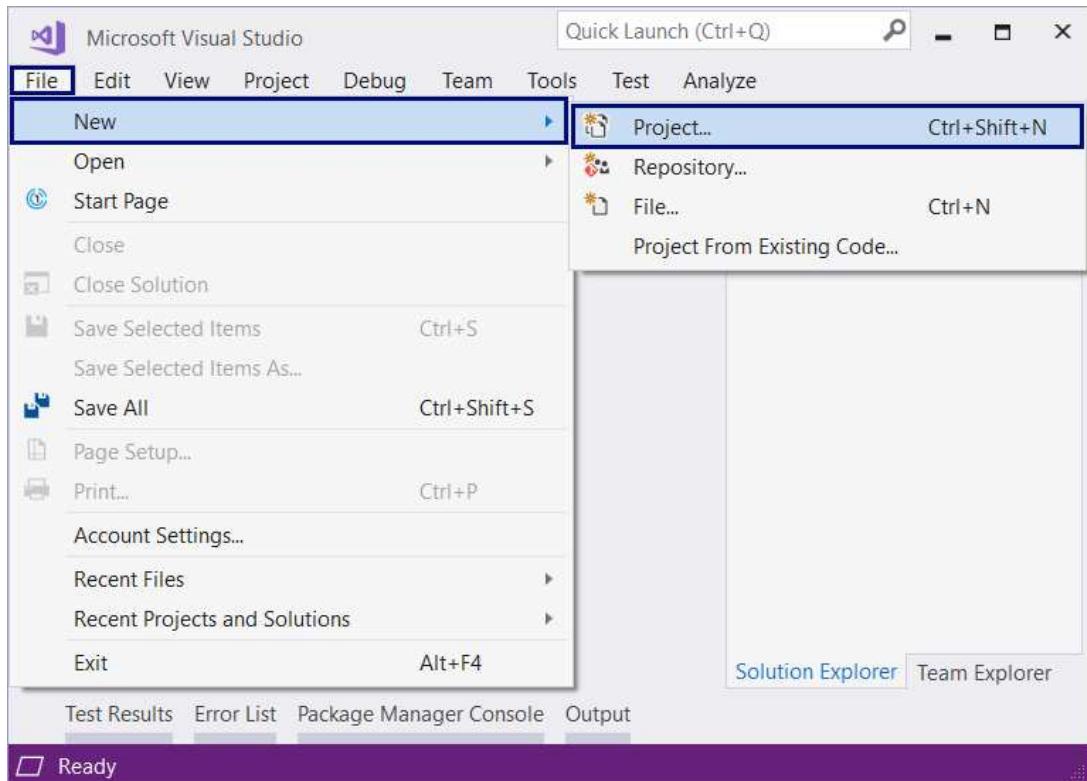
if (a > 100)
{
    Console.WriteLine("The number `a` is bigger than 100");
}
else if (a > 20)
{
    Console.WriteLine("The number `a` is bigger than 20");
}
else
{
    Console.WriteLine("The number `a` is smaller or equal than 20");
}

```

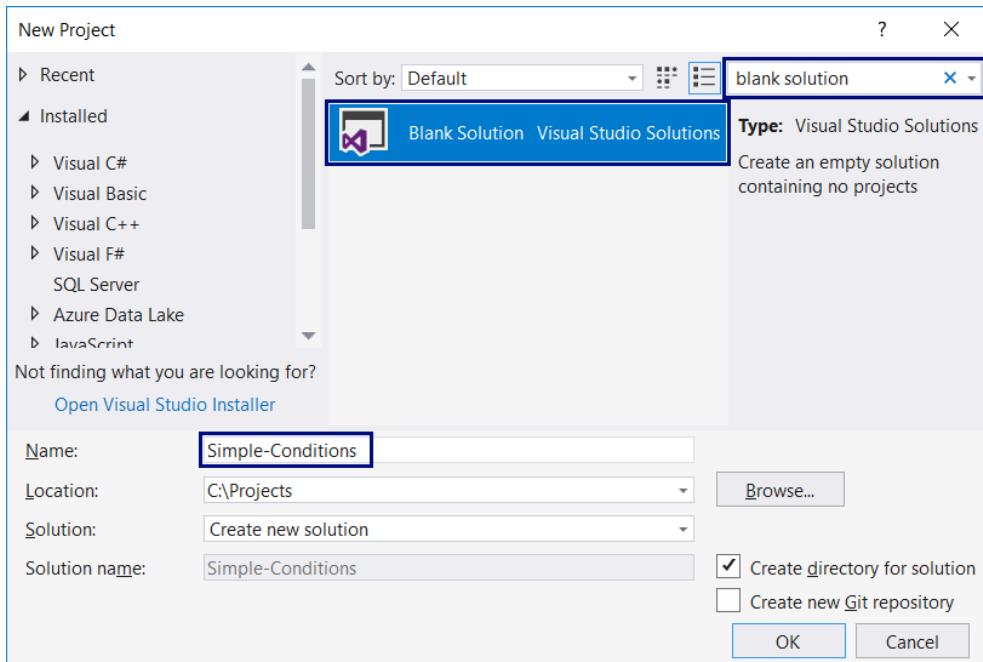
Empty Visual Studio Solution (Blank Solution)

At the start we create a **Blank Solution** in Visual Studio to organize better the task solutions from the exercise – each task will be in a separate project and all projects will be in a common solution.

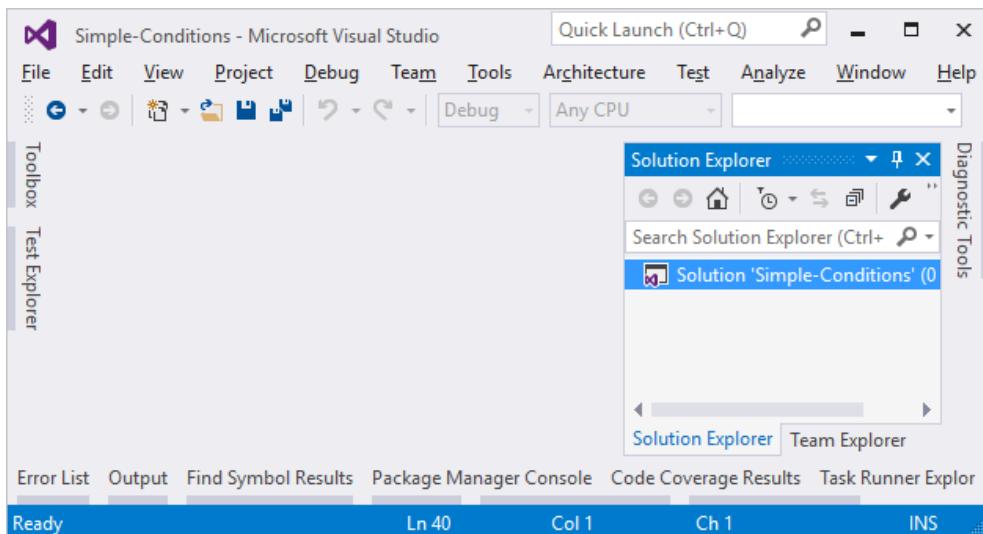
We run Visual Studio and create a new **Blank Solution**: [File] -> [New] -> [Project].



Choose from the dialog box [Templates] -> [Other Project Types] -> [Visual Studio Solutions] -> [Blank Solution] and give an appropriate project name, for example: "Simple-Conditions":



Now we have an **empty Visual Studio Solution** (no projects in it):



We will use this solution to create a **separate project** for each of the problems, which we will solve as exercises in this chapter.

Problem: Excellent Grade

The first exercise for this topic is to write a **console program** that **inputs the grade** (decimal number) and prints **Excellent!** if the grade is **5.50** or higher.

Sample Input and Output

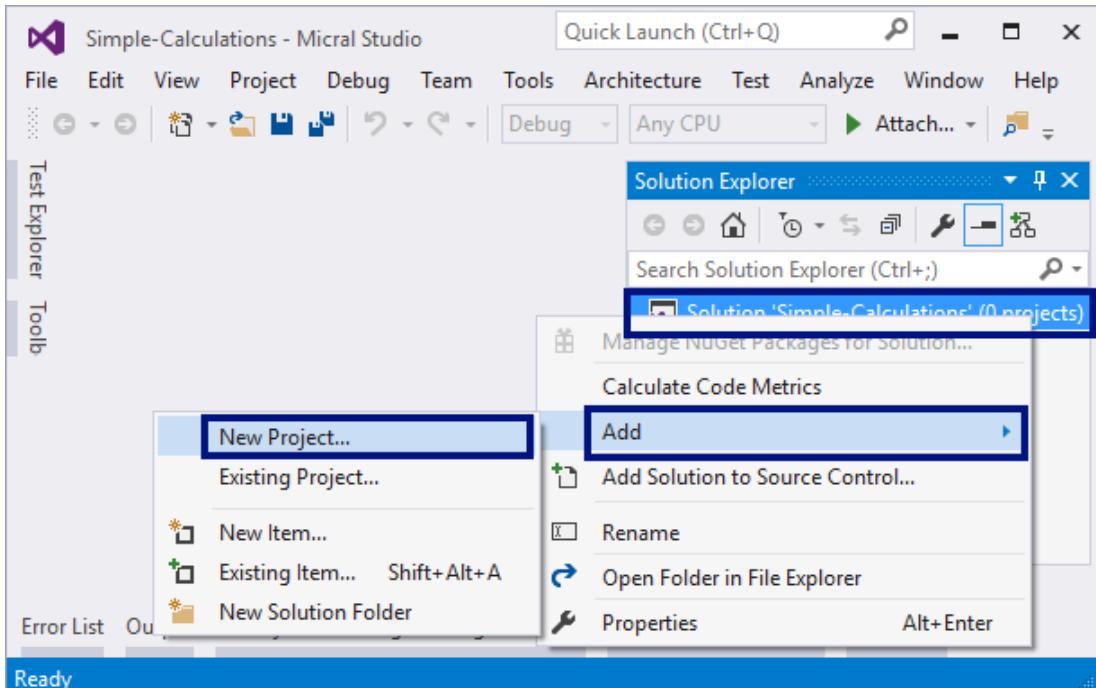
Input	Output
6	Excellent!

Input	Output
5.5	Excellent!

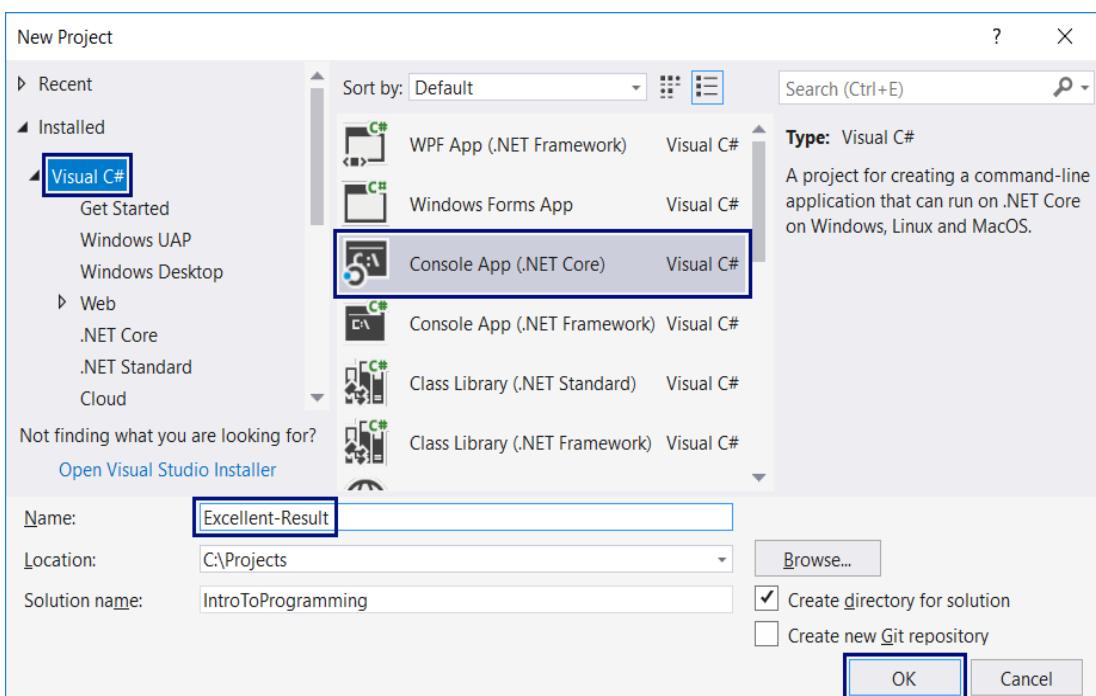
Input	Output
5.49	(no output)

Creating a New C# Project

We create a new project in the existing Visual Studio solution. In **Solution Explorer**, right-click on Solution 'Simple-Conditions'. Then choose [Add] -> [New Project]:



A dialog box will open for selecting a project. Choose **C# console application** and specify a name, for example "Excellent-Result":



Now we have a solution with one console application in it. What remains is to write the code to solve the problem.

Writing the Program Code

For this purpose, we go into the body of the `Main (string[] args)` method and write the following code:

```
namespace Excellent_Result
{
    class Program
    {
        static void Main(string[] args)
        {
            var grade = double.Parse(Console.ReadLine());
            if (grade >= 5.50)
            {
                Console.WriteLine("Excellent!");
            }
        }
    }
}
```

Run the program with [Ctrl+F5], to test it with different input values:

Testing in the Judge System

Test your solution from the example: <https://judge.softuni.org/Contests/Practice/Index/506#0>.

01. Excellent Result

```
1 using System;
2
3 namespace Excellent_Result
4 {
5     class Program
6     {
7         static void Main(string[] args)
8         {
9             var grade = double.Parse(Console.ReadLine());
10            if (grade >= 5.50)
11            {
12                Console.WriteLine("Excellent!");
13            }
14        }
15    }
16 }
```

Allowed working time: 0.100 sec.

Allowed memory: 16.00 MB

Size limit: 16.00 KB

Checker: Case-Insensitive

C# code

Submit

Submissions			
	1		
Points	Time and memory used	Submission date	
✓ 100 / 100	Memory: 7.48 MB Time: 0.031 s	00:12:42 09.04.2019	

Problem: Excellent Grade or Not

The next exercise from this topic is to write a **console program** that **inputs the grade** (decimal number) and prints **Excellent!** if the grade is **5.50** or higher, otherwise "Not excellent.".

Sample Input and Output

Input	Output	Input	Output	Input	Output
6	Excellent!	5	Not Excellent!	5.49	Not excellent.

Creating a New C# Project and Writing the Code

First, we create a new C# console project in the **Simple-Conditions** solution.

- We click on the solution in Solution Explorer and choose [Add] -> [New Project].
- We choose [**Visual C#**] -> [**Windows**] -> [**Console Application**] and specify a name, for example: "Excellent-or-Not".

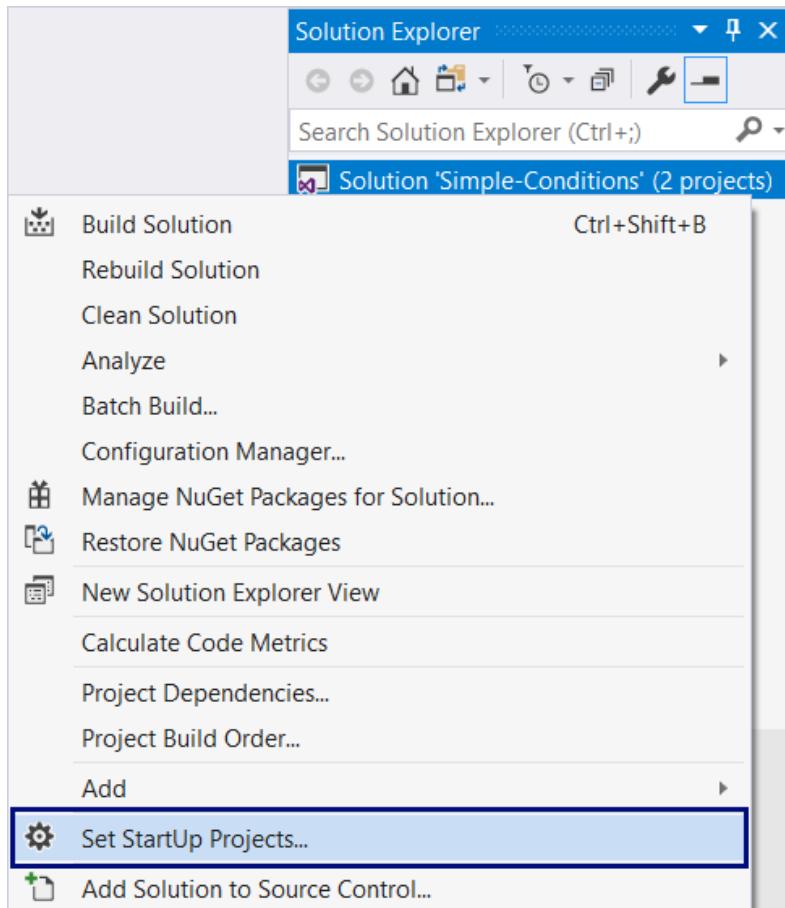
Now we have to **write the code** of the program. You can get help by using the sample code from the picture:

```
var grade = double.Parse(Console.ReadLine());
if (grade >= 5.50)
{
    Console.WriteLine("Excellent!");
}
else
{
    Console.WriteLine("Not excellent.");
}
```

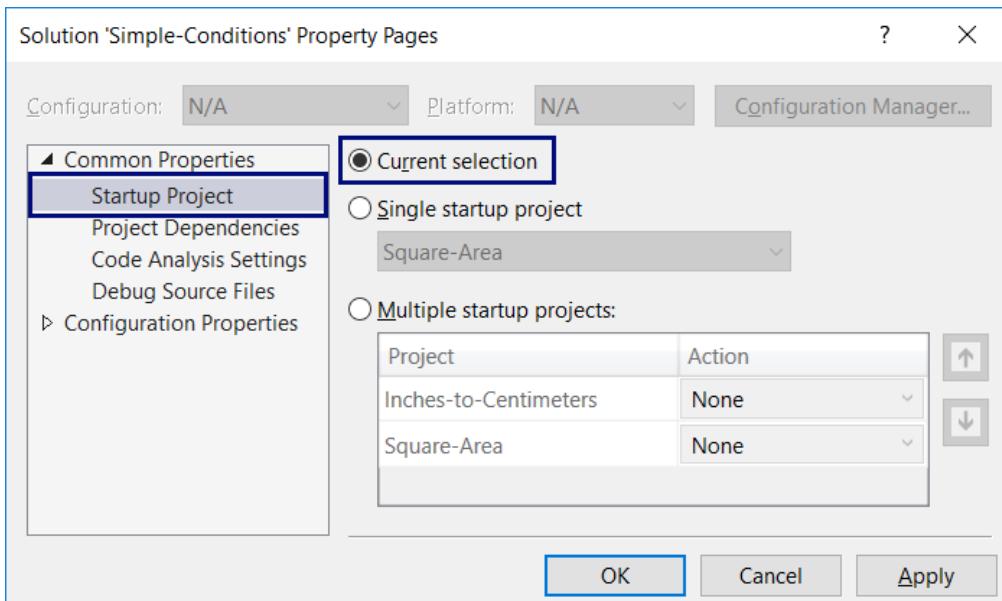
Automatic Switching to the Current Project

If try to run the program using [**Ctrl+F5**], the result might be incorrect, because it might be the result from the previous project, not the current one, shown at the screen.

We turn on the mode for **automatic switching to the current project** by right-clicking on the main Solution and choosing [**Set StartUp Projects...**]:



A dialog box will appear, and you have to choose [Startup Project] -> [Current selection]:



Local Execution and Testing

Now we run the program as usual with [Ctrl + F5] and test if it works correctly:

```
C:\WINDOWS\system32\cmd.e... - □ X
5.6
Excellent!
Press any key to continue . . . -
```

```
C:\WINDOWS\system32\cmd.exe - □ X
4.25
Not excellent.
Press any key to continue . . . -
```

Testing in the Judge System

Test your solution here: <https://judge.softuni.org/Contests/Practice/Index/506#1>.

Submissions		
1	Points	Time and memory used
	100 / 100	Memory: 7.48 MB Time: 0.031 s
	Submission date	12:24:39 09.04.2019
		Details

Problem: Even or Odd

Write a program that checks whether an **integer** is even or odd.

Sample Input and Output

Input	Output	Input	Output	Input	Output	Input	Output
2	even	3	odd	25	odd	1024	even

Hints and Guidelines

Again, first we add a new C# console project in the existing solution. In the **static void Main()** method, we have to write the program code. Checking if a given number is even can be done with the operator **%**, which will return the **remainder from an integer divided by 2** as follows:

```
var isEven = (num % 2 == 0).
```

Now we run the program with [Ctrl + F5] and test it:

```
C:\WINDOWS\system32\...
42
even
Press any key to continue . . .
```

Testing in the Judge System

Test your solution here: <https://judge.softuni.org/Contests/Practice/Index/506#2>.

Problem: Finding the Greater Number

Write a program that inputs **two integers** and prints the larger one.

Sample Input and Output

Input	Output
5	5
3	

Input	Output
3	5
5	

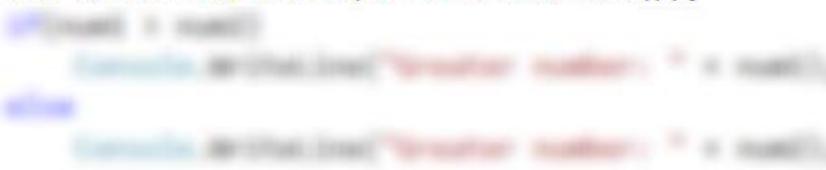
Input	Output
10	10
10	

Input	Output
-5	
5	5

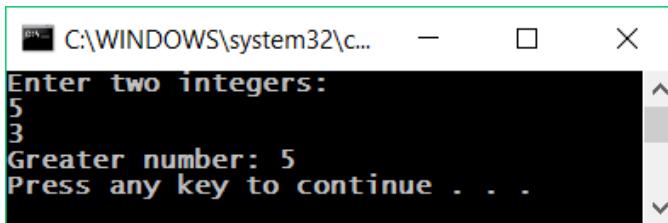
Hints and Guidelines

As usual, first we need to add a **new C# console project** to the existing solution. For the code of the program we need a single **if-else** statement. You can partially get assistance for the code from the picture that is deliberately blurred to make you think about how to write it yourself:

```
Console.WriteLine("Enter two integers:");
var num1 = int.Parse(Console.ReadLine());
var num2 = int.Parse(Console.ReadLine());
```



When we are done with the implementation, so we **run** the program with **[Ctrl + F5]** to test it:



Testing in the Judge System

Test your solution here: <https://judge.softuni.org/Contests/Practice/Index/506#3>.

Problem: Typing a Digit in Words

Write a program that inputs an integer in the range [0 ... 9] and outputs it **in words** in English. If the number is out of range, print "number too big".

Sample Input and Output

Input	Output
5	five

Input	Output
1	one

Input	Output
9	nine

Input	Output
10	number too big

Hints and Guidelines

We can use a series of **if-else** statements to examine the possible **11 cases**.

Testing in the Judge System

Test your solution here: <https://judge.softuni.org/Contests/Practice/Index/506#4>.

Problem: Bonus Score

We have an **integer** – the number of points. **Bonus score** are charged on it, according to the rules described below. Write a program that calculates **bonus score** for this figure and **total points** with bonuses.

- If the number is **up to 100** including, bonus score is 5.
- If the number is **larger than 100**, bonus score is **20%** of the number.
- If the number is **larger than 1000**, bonus score is **10%** of the number.
- Additional bonus score (accrued separately from the previous ones):
 - for **even** number → + 1 p.
 - for number, that **ends with 5** → + 2 p.

Sample Input and Output

Input	Output	Input	Output	Input	Output	Input	Output
20	6	175	37	2703	270.3 2973.3	15875	1589.5 17464.5
26		212					

Video: Bonus Score

Watch the video lesson about the "Bonus Score" problem: <https://youtu.be/hstZ5rNJ7vs>.

Hints and Guidelines

We can calculate the main and additional bonus score with a series of **if-else-if-else** statements. For the **main bonus score** we have **3 cases** (when the entered number is up to 100, between 100 and 1000 and larger than 1000), and for **extra bonus score** – **2 more cases** (when the number is even and odd).

```

var num = int.Parse(Console.ReadLine());
var bonusScore = 0.0;

if (num > 1000)
{
    bonusScore = num * 0.10;
}
else // TODO: Write more logic here...

if (num % 10 == 5)
{
    bonusScore += 2;
}
else // TODO: Write more logic here...

// bonus score:
Console.WriteLine(bonusScore);

// total score:
Console.WriteLine(num + bonusScore);

```

Here's an example of the program execution:

```
C:\WINDOWS\system32...
Enter score: 20
Bonus score: 6
Total score: 26
Press any key to continue . . . -
```

Note that for this exercise, the judge is set to ignore anything that is not a number, so we can print not only the numbers, but also specifying text.

Testing in the Judge System

Test your solution here: <https://judge.softuni.org/Contests/Practice/Index/506#5>.

Problem: Guess the Password

Write a program that **inputs a password** (one line with random text) and checks if the input **matches** the phrase "s3cr3t!P@ssw0rd". If it matches, print "**Welcome**", otherwise "**Wrong password!**".

Sample Input and Output

Input	Output	Input	Output	Input	Output
qwerty	Wrong password!	s3cr3t!P@ssw0rd	Welcome	s3cr3t!p@ss	Wrong password!

Hints and Guidelines

Use an **if-else** statement.

Testing in the Judge System

Test your solution here: <https://judge.softuni.org/Contests/Practice/Index/506#8>.

Problem: Summing Up Seconds

Three athletes finish in a particular number of **seconds** (between **1** and **50**). Write a program that introduces the times of the contestants and calculates their **total time** in "minutes:seconds" format. Seconds need to be **zeroed at the front** (2 -> "02", 7 -> "07", 35 -> "35").

Sample Input and Output

Input	Output	Input	Output	Input	Output	Input	Output
35		22		50		14	
45	2:04	7	1:03	50	2:29	12	0:36
44		34		49		10	

Video: Summing Up Seconds

Watch the video about the "Summing Up Seconds" problem and its step-by-step solution: <https://youtu.be/bnNxe79X-wo>.

Hints and Guidelines

First, we sum up the three numbers to get the total result in seconds. Since **1 minute = 60** seconds, we will have to calculate the number of minutes and seconds in the range 0 to 59:

- If the result is between 0 and 59, we print 0 minutes + calculated seconds.
- If the result is between 60 and 119, we print 1 minute + calculated seconds minus 60.
- If the result is between 120 and 179, we print 2 minutes + calculated seconds minus 120.
- If the seconds are less than 10, we print the number with zero in front.

```
int firstCompetitor = int.Parse(Console.ReadLine());
// TODO: Read also second and third competitors' time

int seconds = firstCompetitor + secondCompetitor + thirdCompetitor;
int minutes = 0;

if (seconds > 59)
{
    minutes++;
    seconds = seconds - 60;
}

if (seconds > 59)
{
    minutes++;
    seconds = seconds - 60;
}

if (seconds < 10)
{
    Console.WriteLine(minutes + ":" + "0" + seconds);
}
else
{
    Console.WriteLine(minutes + ":" + seconds);
}
```

Testing in the Judge System

Test your solution here: <https://judge.softuni.org/Contests/Practice/Index/506#6>.

Problem: Metric Converter

Write a program that convert a distance between the following 8 units of measure: **m, mm, cm, mi, in, km, ft, yd**. Use the below table:

Input measure	Output measure	Input measure	Output measure
1 meter (m)	1000 millimeters (mm)	1 meter (m)	0.001 kilometers (km)
1 meter (m)	100 centimeters (cm)	1 meter (m)	3.2808399 feet (ft)
1 meter (m)	0.000621371192 miles (mi)	1 meter (m)	1.0936133 yards (yd)
1 meter (m)	39.3700787 inches (in)		

You have three **input** lines:

- First line: the number for converting.
- Second line: the input unit.
- Third line: the output unit (for result).

Sample Input and Output

Input	Output	Input	Output	Input	Output
12 km ft	39370.0788	150 mi in	9503999.99393599	450 yd km	0.41147999937455

Video: Metric Converter

Watch the following video lesson to learn how to solve the "Metric Converter" problem in C#:
<https://youtu.be/Bd6NgaHhrko>.

Hints and Guidelines

We read the input data, and we can add **ToLower()** function when we read the measuring units. The function will make all letters small. As we can see from the table in the condition, we can only do converting **between meters and some other measuring unit**. Then, first we have to calculate the number for converting in meters. That's why, we need to make a set of checks to define the input unit and then the output unit.

```
var size = double.Parse(Console.ReadLine());
var sourceMetric = Console.ReadLine().ToLower();
var destMetric = Console.ReadLine().ToLower();
if (sourceMetric == "km")
{
    size = size / 0.001;
}
// Check the other metrics: mm, cm, ft, yd, ...
if (destMetric == "ft")
{
    size = size * 3.2808399;
}
// Check the other metrics: mm, cm, ft, yd, ...
Console.WriteLine(size + " " + destMetric);
```

Testing in the Judge System

Test your solution here: <https://judge.softuni.org/Contests/Practice/Index/506#7>.

Problem: Numbers from 100 to 200

Write a program that **inputs an integer** and checks if it is **below 100, between 100 and 200 or over 200**. Print the appropriate message as in the examples below.

Sample Input and Output

Input	Output	Input	Output	Input	Output
95	Less than 100	120	Between 100 and 200	210	Greater than 200

Testing in the Judge System

Test your solution here: <https://judge.softuni.org/Contests/Practice/Index/506#9>.

Problem: Identical Words

Write a program that **inputs two words** and checks if they are the same. Do not make difference between uppercase and lowercase letters. You have to print "yes" or "no".

Sample Input and Output

Input	Output	Input	Output	Input	Output	Input	Output
Hello Hello	yes	SoftUni softuni	yes	Soft Uni	no	banana lemon	no

Hints and Guidelines

Before comparing the words, turn them into a lowercase to avoid the letter size influence (uppercase / lowercase): `word = word.ToLower()`.

Testing in the Judge System

Test your solution here: <https://judge.softuni.org/Contests/Practice/Index/506#10>.

Problem: Speed Assessment

Write a program that **inputs the speed** (decimal number) and prints **speed information**. For speed **up to 10** (inclusive), print "slow". For speed **over 10** and **up to 50**, print "average". For speed **over 50** and **up to 150**, print "fast". For speed **over 150** and **up to 1000**, print "ultra fast". For higher speed, print "extremely fast".

Sample Input and Output

Input	Output	Input	Output	Input	Output
8	slow	126	fast	3500	extremely fast
49.5	average	160	ultra fast	50000	extremely fast

Testing in the Judge System

Test your solution here: <https://judge.softuni.org/Contests/Practice/Index/506#11>.

Problem: Areas of Figures

Write a program that **inputs the sizes of a geometric figure** and **calculates its area**. The figures are four types: **square**, **rectangle**, **circle** and **triangle**.

The first line of the input provides the type of the figure (**square**, **rectangle**, **circle**, **triangle**).

- If the figure is a **square**, the next line provides one number – the length of its **side**.
- If the figure is a **rectangle**, the next two lines provide two numbers – the lengths of its **sides**.
- If the figure is a **circle**, the next line provides one number – the **radius** of the circle.
- If the figure is a **triangle**, the next two lines provide two numbers – the length of the **side** and the length of its **height**.

Round the result up to the **third digit after the decimal point**.

Sample Input and Output

Input	Output	Input	Output	Input	Output	Input	Output
square 5	25	rectangle 7 2.5	17.5	circle 6	113.097	triangle 4.5 20	45

Testing in the Judge System

Test your solution here: <https://judge.softuni.org/Contests/Practice/Index/506#12>.

Problem: Time + 15 Minutes

Write a program that **inputs hours and minutes** of a 24-hour day and calculates what will be the time **after 15 minutes**. Print the result in **hh:mm** format. Hours are always between 0 and 23, and minutes are always between 0 and 59. Hours are written with one or two digits. Minutes are always written with two digits and zero at the front when needed.

Sample Input and Output

Input	Output	Input	Output	Input	Output	Input	Output
1 46	2:01	0 01	0:16	23 59	0:14	11 08	11:23

Hints and Guidelines

Add 15 minutes and check using a few conditions. If minutes are over 59, **increase hours** with 1 and **reduce minutes** with 60. Identically, check the case when hours are over 23. When you print the minutes, you should **check for zero at the front**.

Testing in the Judge System

Test your solution here: <https://judge.softuni.org/Contests/Practice/Index/506#13>.

Problem: Equal 3 Numbers

Write a program that **inputs 3 numbers** and prints whether they are the same (**yes / no**).

Sample Input and Output

Input	Output	Input	Output	Input	Output
5 5 5	yes	5 4 5	no	1 2 3	no

Testing in the Judge System

Test your solution here: <https://judge.softuni.org/Contests/Practice/Index/506#14>.

Problem: * Numbers from 0 to 100 as English Words

Write a program that converts a number in the range of [0 ... 100] into text (in English).

Sample Input and Output

Input	Output	Input	Output	Input	Output
25	twenty five	42	forty two	6	six

Hints and Guidelines

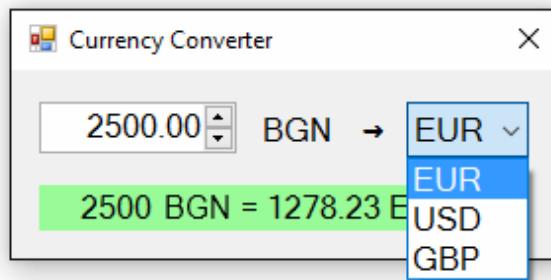
First check for **one-digit numbers** and if the number is one-digit, print the appropriate word for it. Then check for **two-digit numbers**. Print them in two parts: left part (**tens** = number / 10) and right part (**units** = number % 10). If the number has 3 digits, it must be 100 and can be considered a special case.

Testing in the Judge System

Test your solution here: <https://judge.softuni.org/Contests/Practice/Index/506#15>.

Lab: GUI (Desktop) Application – Currency Converter

After we've done some exercises on **conditional statements (checks)**, now let's do something more interesting: an application with a graphical user interface (**GUI**) for converting currencies. We will use the knowledge from this chapter to choose from several available currencies and make calculations at different rate to the selected currency. Now let's see how to create a graphical (**GUI**) app for **currency conversion**. The app will look like the picture below:



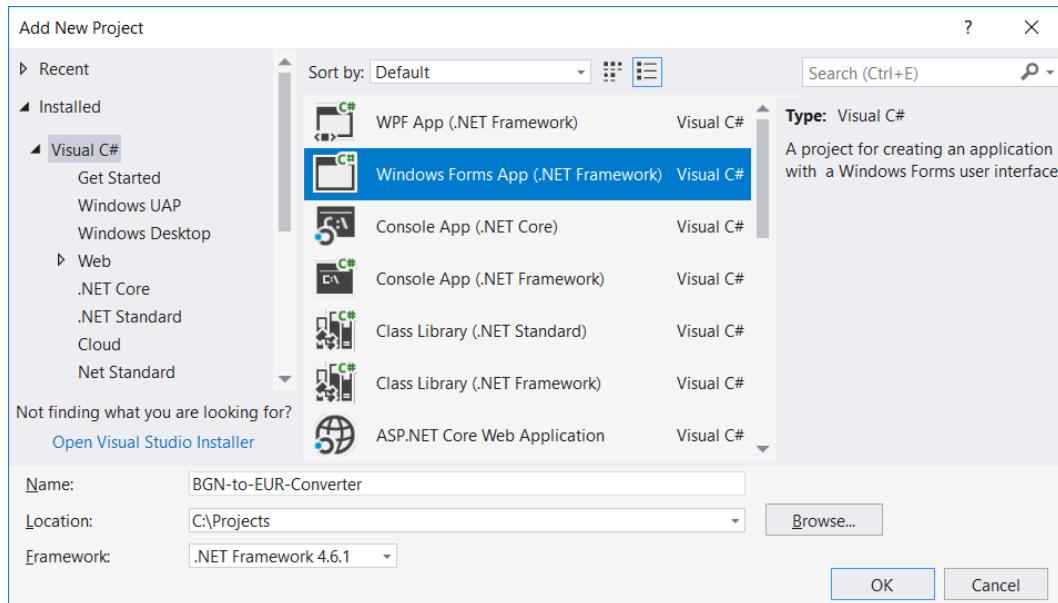
It converts Bulgarian levs (BGN) to Euro (EUR), US Dollars (USD) or Great Britain Pounds (GBP).

Video: Building a GUI App "Currency Converter"

Watch the video lesson about building a Windows Forms based GUI app "Currency Converter": <https://youtu.be/lIkPmoXmjdg>.

Creating a New C# Project and Adding Controls

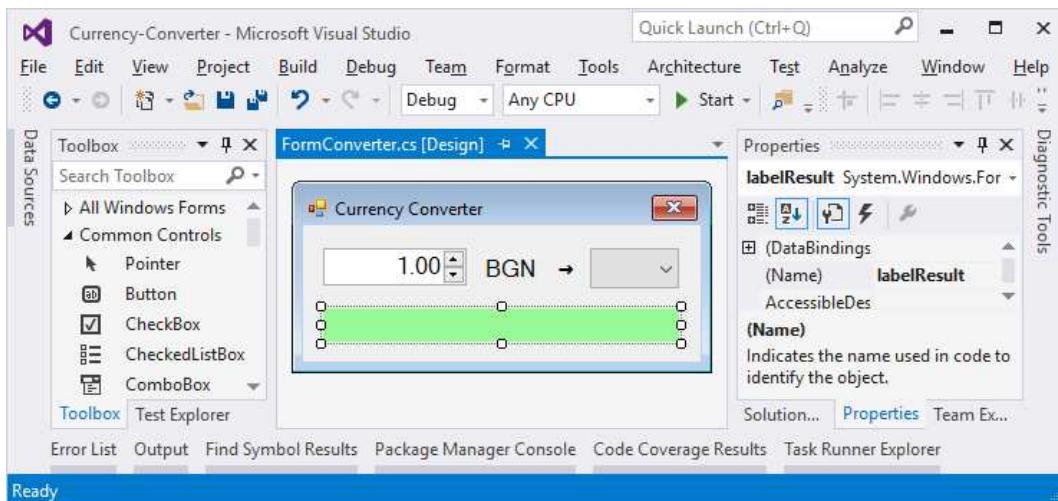
This time we create a new **Windows Forms Application** with name "Currency-Converter":



We order the following controls in the form:

- One box for entering a number (**NumericUpDown**)
- One drop-down list with currencies (**ComboBox**)
- Text block for the result (**Label**)
- Several inscriptions (**Label**)

We set the **sizes** and their properties to look like the picture below:



Configuring the UI Controls

We apply the following **settings** for the UI controls:

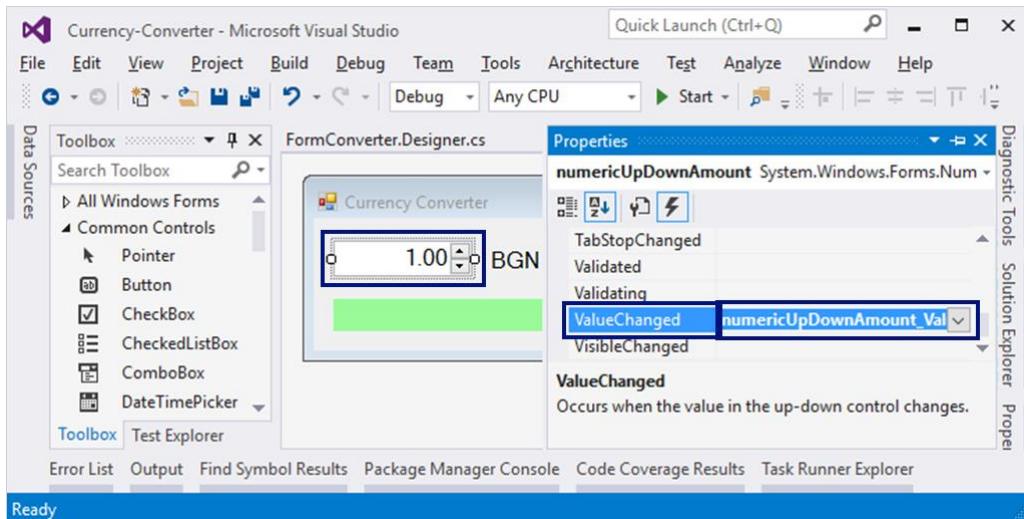
- For the main form (**Form**) that contains all the controls:
 - **(name)** = **FormConverter**
 - **Text** = "Currency Converter"
 - **Font.Size** = 12
 - **MaximizeBox** = False
 - **MinimizeBox** = False

- `FormBorderStyle = FixedSingle`
- For the field for entering a number (`NumericUpDown`):
 - `(name) = numericUpDownAmount`
 - `Value = 1`
 - `Minimum = 0`
 - `Maximum = 1000000`
 - `TextAlign = Right`
 - `DecimalPlaces = 2`
- For the drop-down list of currencies (`ComboBox`):
 - `(name) = comboBoxCurrency`
- `DropDownStyle = DropDownList`
 - `Items =`
 - EUR
 - USD
 - GBP
- For the result text block (`Label`):
 - `(name) = labelResult`
 - `AutoSize = False`
 - `BackColor = PaleGreen`
 - `TextAlign = MiddleCenter`
 - `Font.Size = 14`
 - `Font.Bold = True`

Events and Event Handlers

We need to take the following **events** to write the C# code that will be executed upon their occurrence:

- The event **ValueChanged** of numeric entry control `numericUpDownAmount`:



- The event **Load** of the form `FormConverter`
- The event **SelectedIndexChanged** of the drop-down list for choosing the currency `comboBoxCurrency`

We will use the following **C# code** for event handling:

```
private void FormConverter_Load(object sender, EventArgs e)
{
    this.comboBoxCurrency.SelectedItem = "EUR";
}

private void numericUpDownAmount_ValueChanged(object sender, EventArgs e)
```

```

{
    ConvertCurrency();
}

private void comboBoxCurrency_SelectedIndexChanged(object obj, EventArgs e)
{
    ConvertCurrency();
}

```

Our task is to select the currency "EUR" when we start the program and change the values in the sum or currency field then calculating the result by calling the **ConvertCurrency()** method.

Writing the Program Code

We have to write the event **ConvertCurrency()** to convert the BGN amount into the selected currency:

```

private void ConvertCurrency()
{
    var originalAmount = this.numericUpDownAmount.Value;
    var convertedAmount = originalAmount;

    if (this.comboBoxCurrency.SelectedItem.ToString() == "EUR")
    {
        convertedAmount = originalAmount / 1.95583m;
    }
    else if (this.comboBoxCurrency.SelectedItem.ToString() == "USD")
    {
        convertedAmount = originalAmount / 1.80810m;
    }
    else if (this.comboBoxCurrency.SelectedItem.ToString() == "GBP")
    {
        convertedAmount = originalAmount / 2.54990m;
    }
    this.labelResult.Text = originalAmount + " BGN = " +
        Math.Round(convertedAmount, 2) + " " +
        this.comboBoxCurrency.SelectedItem;
}

```

The above code takes **the amount** for converting the field **numericUpDownAmount** and **the selected currency** for the result from the field **comboBoxCurrency**. Then with a **conditional statement**, according to the selected currency, the amount is divided by **the exchange rate** (which is fixed in the source code). Finally, a text **message with the result** (rounded to the second digit after the decimal point) is generated and recorded in the green box **labelResult**. Try it!

If you have problems with the example above, you can ask for help in the SoftUni official **discussion forum** (<http://forum.softuni.org>) or in the SoftUni official **Facebook page** (<https://fb.com/softuni.org>).

High-quality education profession and job for software engineers

- ✓ Free start for **absolute beginners**
- ✓ Choose among the **most in-demand** professions in the software industry
- ✓ Assistance for **career start**
- ✓ **Top trainers**, mentors and active learning **community**



SoftUni provides practical, modern and **innovative education** for programming, IT, design, marketing and digital skills for thousands of young people. The "**Software University**" program builds true **software engineering professionals**.

Join the end-to-end **software engineering learning program** at SoftUni to master the most in-demand programming concepts, languages, software technologies and skills through a modern teaching methodology, **interactive learning platform** and tons of practical exercises and projects. Our **curriculum** is developed together with the **IT companies** to closely match the industry demands.

SoftUni works directly with the **companies** from the software industry to **find a job** for its students and to make them successful software engineers.

The path of the student at SoftUni



Apply now

softuni.org/apply

Chapter 3.2. Simple Conditions – Exam Problems

In the previous chapter, we went through the simple conditional statements in C#, which we can use to execute different actions depending on a given condition. Now we shall solve a few **practical exercises** from SoftUni exams to gain some experience.

Simple Conditions – Quick Review

We mentioned what **the scope** of a variable is, and how to track the execution of our program step by step (the so-called **debugging**) as well. In this chapter, we will practice working with **simple conditions** by going through some exam tasks. To do this, let's first revise the **if-else** construction:

```
if (bool expression)
{
    // condition body;
}
else
{
    // else-construction body;
}
```

if conditions in C# consist of:

- **if clause**
- **bool expression** – a variable of bool type (**bool**) or bool logical expression (an expression that results in **true/false**)
- condition **body** – contains random block of source code
- **else clause** and its block of source code (**optional**)

After having revised how to write simple conditions, let's solve a few **exam problems** in order to practice the **if-else** construction.

Problem: Transportation Price

A student has to travel **n kilometers**. He can choose between **three types of transportation**:

- **Taxi**. Starting fee: **0.70** EUR. Day rate: **0.79** EUR/km. Night rate: **0.90** EUR/km.
- **Bus**. Day / Night rate: **0.09** EUR/km. Can be used for distances of minimum **20** km.
- **Train**. Day / Night rate: **0.06** EUR/km. Can be used for distances of minimum **100** km.

Write a program that reads the number of **kilometers n** and **period of the day** (day or night) and calculates **the price for the cheapest transport**.

Input Data

Two lines are read from the console:

- The first line holds a number **n** – number of kilometers – an integer in the range of **[1 ... 5000]**.
- The second line holds the word “**day**” or “**night**” – traveling during the day or during the night.

Output Data

Print on the console **the lowest price** for the given number of kilometers.

Sample Input and Output

Input	Output	Input	Output	Input	Output	Input	Output
5 day	4.65	7 night	7	25 day	2.25	180 night	10.8

Hints and Guidelines

We will read the input data and depending on the distance, we will choose the cheapest transport. To do that, we will write a few conditional statements.

Processing the Input Data

In the task, we are given **information about the input and output data**. Therefore, in the first two lines from the solution, we will declare and initialize the two **variables** that are going to store **the values of the input data**. The first line contains **an integer** and that is why the declared variable will be of **int** type. The second line contains **a word**, therefore, the variable will be of **string** type.

```
int distance = int.Parse(Console.ReadLine());
string dayOrNight = Console.ReadLine();
```

Before starting with the conditional statements, we need to **declare** a **variable** that stores the value of **the transport price**.

```
double price = 0;
```

Calculating Taxi Rate

After having **declared and initialized** the input data and the variable that stores the value of the price, we have to decide which **conditions** of the task have to be **checked first**.

The task specifies that the rates of two of the vehicles **do not depend** on whether it is **day or night**, but the rate of one of the transports (taxi) **depends**. This is why the **first condition** will be whether it is **day or night**, so that it is clear which rate the taxi will be **using**. To do that, we **declare one more variable** that stores **the value of the taxi rate**.

```
double taxiRate = 0;
```

In order to calculate **the taxi rate**, we will use conditional statement of type **if-else** and through it, the variable for the price of the taxi will store its **value**.

```
if (dayOrNight == "day")
    taxiRate = 0.79;
else
    taxiRate = 0.90;
```

Calculating Transportation Price

After having done that, now we can start calculating **the transport price** itself. The constraints in the task refer to **the distance** that the student wants to travel. This is why, we will use an **if-else** statement that will help us find **the price** of the transport, depending on the given kilometers.

```
if (distance < 20)
    price = 0.70 + distance * taxiRate;
else if (distance < 100)
    price = distance * 0.09;
```

```
else
    price = distance * 0.06;
```

First, we check whether the kilometers are **less than 20**, as the task specifies that the student can only use **a taxi** for **less than 20 kilometers**. If the condition is **true** (returns **true**), the variable that is created to store the value of the transport (**price**), will **store** the corresponding value. This value equals **the starting fee** that we will sum with its **rate**, **multiplied by the distance** that the student has to travel.

If the condition of the variable **is not true** (returns **false**), the next step of our program is to check whether the kilometers are **less than 100**. We do that because the task specifies that in this range, **a bus** can be used as well. **The price** per kilometer of a bus **is cheaper** than a taxi one. Therefore, if the result of the condition is **true**, we store **a value**, equal to the result of the **multiplication of the rate** of the bus by **the distance** to the variable for the transportation **price** in the **else if** statement body.

If this condition **does not return true** as a result, we have to store **a value**, equal to **the result of the multiplication of the distance by the train rate** to the price variable in the **else** body. This is done because the train is **the cheapest** transport for the given distance.

Printing the Output Data

After we have checked the distance **conditions** and we have **calculated the price of the cheapest transport**, we have to **print it**. The task does not specify how to format the result, therefore, we just print **the variable**:

```
Console.WriteLine(price);
```

Testing in the Judge System

Test your solution here: <https://judge.softuni.org/Contests/Practice/Index/507#0>.

Problem: Pipes in Pool

A pool with **volume V** fills up via **two pipes**. Each pipe has a certain **flow rate** (the liters of water, flowing through a pipe for an hour). A worker starts the pipes simultaneously and goes out for **N hours**. Write a program that finds the state of the pool **the moment the worker comes back**.

Input Data

Four lines are read from the console:

- The first line contains a number **V** – the **volume of the pool in liters** – an integer in the range of **[1 ... 10000]**.
- The second line contains a number **P1** – the **flow rate of the first pipe per hour** – an integer in the range of **[1 ... 5000]**.
- The third line contains a number **P2** – the **flow rate of the second pipe per hour** – an integer in the range of **[1 ... 5000]**.
- The fourth line contains a number **H** – the **hours that the worker is absent** – a floating-point number in the range of **[1.0 ... 24.00]**.

Output Data

Print on the console **one of the two possible states**:

- To what extent the pool has filled up and how many percent each pipe has contributed with. All percent values must be formattted to an integer (without rounding).

- "The pool is [x]% full. Pipe 1: [y]%. Pipe 2: [z]."
- If the pool has overflowed – with how many liters it has overflowed for the given time – a floating-point number.
 - "For [x] hours the pool overflows with [y] liters."

Have in mind that due to **the rounding to an integer**, there is **data loss** and it is normal **the sum of the percents to be 99%, not 100%**.

Sample Input and Output

Input	Output	Input	Output
1000 100 120 3	The pool is 66% full. Pipe 1: 45%. Pipe2: 54%.	100 100 100 2.5	For 2.5 hours the pool overflows with 400 liters.

Hints and Guidelines

In order to solve the task, we read the input data, write a few conditional statements, do some calculations and print the result.

Processing the Input Data

From the task requirements we note that our program must have **four lines** from which we read **the input data**. The first **three** consist of **integers** and that is why the **variables** that will store their values will be of **int** type. We know that the **fourth** line will be a **floating-point number**, therefore, the variable we use will be of **double** type.

```
int volume = [ ]
int pipe1 = [ ]
int pipe2 = [ ]
double hours = [ ]
```

Our next step is to **declare and initialize** a variable in which we are going to calculate with how many **liters** the pool has **filled up** for the time the worker was **absent**. We do the calculations by **summing** the values of the flow rates of the **two pipes** and **multiplying** them by the **hours** that are given as input data.

```
double water = [ ]
```

Checking the Conditions and Processing Output Data

After we have **the value of the quantity** of water that has flown through the **pipes**, the next step is to **compare** that quantity with the volume of the pool itself.

We do that with a simple **if-else** statement, where the condition will be whether **the quantity of water is less than the volume of the pool**. If the statement returns **true**, we have to print **one** line that contains **the ratio** between the quantity of **water that has flown through the pipes** and **the volume of the pool**, as well as the **ratio of the quantity of the water from each pipe to the volume of the pool**.

The ratio has to be in **percentage**, that is why all the calculations so far will be **multiplied by 100**. The values will be printed using **placeholders**, and as there is a condition **the result in percentage** to be

formatted to **two digits** after the decimal point **without rounding**, we will use the method `Math.Truncate(...)`.

```
if (water <= volume)
{
    Console.WriteLine("The pool is {0}% full. Pipe 1: {1}%. Pipe 2: {2}%", 
        Math.Truncate(...),
        Math.Truncate(...),
        Math.Truncate(...));
}
else
{
    Console.WriteLine("For {0} hours the pool overflows with {1} liters.", 
        hours, water - volume);
}
```

However, if the condition returns `false`, that means that the quantity of water is more than the volume of the pool, therefore, it has **overflowed**. Again, the output data has to be on **one line**, but this time it should contain only **two values** – the one of the **hours** when the worker was absent, and the quantity of water, which is the **difference** between the incoming water and the volume of the pool.

Testing in the Judge System

Test your solution here: <https://judge.softuni.org/Contests/Practice/Index/507#1>.

Problem: Sleepy Tom Cat

Tom Cat likes to sleep all day but, unfortunately, his owner is always playing with him whenever he has free time. In order to sleep well, the norm of games that Tom has is 30 000 minutes per year. The time for games he has depends on the holidays that his owner has:

- During **workdays**, his owner plays with him **63 minutes per day**.
- During **holidays**, his owner plays with him **127 minutes per day**.

Write a program that reads the number of holidays and prints whether Tom can sleep well and how much the difference from the norm for the current year is. It is assumed that there are 365 days in one year.

Example: 20 holidays -> the working days are 345 ($365 - 20 = 345$). The time for games is 24 275 minutes ($345 * 63 + 20 * 127$). The difference from the norm is 5 725 minutes ($30\,000 - 24\,275 = 5\,725$) or 95 hours and 25 minutes.

Input Data

The input is read from the console and consists of an integer – the number of holidays in the range of [0 ... 365].

Output Data

Two lines have to be printed on the console:

- If Tom's time for games is above the norm for the current year:
 - On the first line print: “Tom will run away”
 - On the second line print the difference from the norm in the format: “{H} hours and {M} minutes more for play”

- If the time for games of Tom is below the norm for the current year:
 - On the first line print: "Tom sleeps well"
 - On the second line print the difference from the norm in the format: "{H} hours and {M} minutes less for play"

Sample Input and Output

Input	Output
20	Tom sleeps well 95 hours and 25 minutes less for play

Input	Output
113	Tom will run away 3 hours and 47 minutes for play

Hints and Guidelines

In order to solve the problem, we will read the input data. Then, we will write a few conditional statements and do some calculations. Finally, we will print the result.

Reading the Input Data

From the task we see that the input data will be read only on the first line and will be an integer in the range of [0 ... 365]. This is why we will use a variable of int type.

```
var holidays =
```

Calculating Working Days

To solve the problem, first we have to calculate the total minutes the owner of Tom is playing with him. We see that not only does the sleepy cat has to play with his owner during the holidays, but also during the working days. The number that we read from the console refers to the holidays.

Our next step is to calculate, with the help of that number, how many the working days of the owner are, as without them we cannot calculate the total minutes for play. As the total number of days per year is 365 and the number of holidays is X, that means that the number of working days is $365 - X$. We store the difference in a new variable that only stores this value.

```
var workingDays =
```

Calculating Playing Time

Once we have the number of days for playing, we can calculate the time for games of Tom in minutes. Its value is equal to the result of the multiplication of the working days by 63 minutes (the task specifies that during working days, the time for play is 63 minutes per day), summed with the result of the multiplication of the holidays by 127 minutes (the task specifies that during holidays, the time for play is 127 minutes per day).

```
var totalPlayMinutes =
```

In the task condition we see that we have to print the difference between the two values in hours and minutes as output data. That is why we subtract the total time for play from the norm of 30 000 minutes and store the result in a new variable. After that, we divide that variable by 60 to get the hours, and then, to find out how many the minutes are, we use modular division with the operator %, as again we divide the variable of the difference by 60.

Here we have to note that if the total time for play of Tom is less than 30,000, when subtracting the norm from it, we will obtain a negative number. In order to neutralize the number in the division, we

use **the method `Math.Abs(...)`** when finding the difference. The code below gives an idea of how to implement this:

```
var difference = Math.Abs( _____ );
var hours =
var minutes =
```

Checking the Conditions

The time for games is already calculated, which leads us to the **next step – comparing the time for play of Tom with the norm** on which the good sleep of the cat depends. For that we will use an **if-else** conditional statement. In the **if clause** we will check whether **the time for play is more than 30 000** (the norm).

Processing the Output Data

Whatever the **result** of the conditional statement is, we have to print how much **the difference in hours and minutes** is. We will do that with a **placeholder** and the variables that store the values of the hours and the minutes, as the formatting will be according to the task requirements for output.

```
if (totalPlayMinutes > 30000)
{
    Console.WriteLine("Tom will run away");
    Console.WriteLine("{0} hours and {1} minutes more for play",
        hours, minutes);
}
else
{
    Console.WriteLine("Tom sleeps well");
    Console.WriteLine("{0} hours and {1} minutes less for play",
        hours, minutes);
}
```

Testing in the Judge System

Test your solution here: <https://judge.softuni.org/Contests/Practice/Index/507#2>.

Problem: Harvest

In a vineyard with area X square meters, 40% of the harvest goes for wine production. Y kilograms of grapes are extracted from 1 m² vineyard. 2,5 kg of grapes are needed for 1 liter of wine. The wanted quantity of wine for sale is Z liters.

Write a program that calculates how much wine can be produced and whether that quantity is enough. If it is enough, the rest is divided between the vineyard workers equally.

Input Data

The input data is read from the console and consists of **exactly 4 lines**:

- First line: X m² is the vineyard – an integer in the range of [10 ... 5000].
- Second line: Y grapes for one m² – an integer in the range of [0.00 ... 10.00].
- Third line: Z needed liters of wine – an integer in the range of [10 ... 600].

- Fourth line: **number of workers** – an integer in the range of [1 ... 20].

Output Data

The following has to be printed on the console:

- If the produced wine is **less than the needed quantity**:
 - “It will be a tough winter! More {insufficient wine} liters wine needed.”
* The result has to be rounded down to the nearest integer.
- If the produced wine is **more than the needed quantity**:
 - “Good harvest this year! Total wine: {total wine} liters.”
* The result has to be rounded down to the nearest integer.
 - “{Wine left} liters left -> {wine for one worker} liters per person.”
* Both of the results have to be rounded up to the higher integer.

Sample Input and Output

Input	Output
650 2 175 3	Good harvest this year! Total wine: 208 liters. 33 liters left -> 11 liters per person.

Input	Output
1020 1.5 425 4	It will be a tough winter! More 180 liters wine needed.

Hints and Guidelines

In order to solve the problem, we will read the input data. Then, we will write a few conditional statements and do some calculations. Finally, we will print the result.

Processing the Input Data

First, we have to **check what the input data** will be, so that we can choose what **variables** we will use.

```
var vineyardArea = [REDACTED]
var grapePerSquare = [REDACTED]
var neededLiters = [REDACTED]
var workers = [REDACTED]
```

Performing the Calculations

To solve the problem, based on **the input data**, we have to **calculate** how many **liters of wine** will be produced. From the task requirements, we see that in order to **calculate** the quantity of **wine in liters**, we firstly have to find **the quantity of grapes in kilograms**, which will be get from the harvest. For that, we will **declare a variable** that keeps a **value**, equal to **40%** of the result from the **multiplication** of the vineyard area by the quantity of grapes, which is extracted from 1 m^2 .

After having done these calculations, we are ready to **calculate the quantity of wine in liters** that will be produced from the harvest as well. For that, we **declare** one more **variable** that stores that **quantity**, which in order to calculate, we have to **divide the quantity of grapes in kg by 2.5**.

```
var harvestPerVine = [REDACTED]
var vine = [REDACTED]
```

Checking the Conditions and Printing the Output

After having done the necessary calculations, the next step is to check whether the liters of wine that have been produced, are enough. For that we will use a simple conditional statement of the **if-else** type and we will check whether the liters of wine from the harvest are more than or equal to the needed liters.

If the condition returns **true**, from the requirements we see that on the first line we have to print the wine that has been produced from the harvest. That value has to be rounded down to the nearest integer, which we will do by using the method **Math.Floor(...)** and a placeholder when printing it.

On the second line we have to print the results by rounding them up to the higher integer, which we will do by using the method **Math.Ceiling(...)**. The values that we have to print are the quantity of wine left and the quantity that each worker gets. The wine left is equal to the difference between the produced liters of wine and the needed liters of wine. We calculate the value of that quantity in a new variable, which we declare and initialize in the **if condition body**, before printing the first line. We calculate the quantity of wine that each worker gets by dividing the wine left by the number of workers.

```
if ( )
{
    var vineyardsLeft =
        Console.WriteLine("Good harvest this year! Total wine: {0} liters.",
            Math.Floor( ));

    Console.WriteLine("{0} liters left -> {1} liters per person.",
        Math.Ceiling( ),
        Math.Ceiling( ));
}
```

If the condition returns **false**, we have to print the difference between the needed liters and the liters of wine produced from the harvest. There is a specification that the result has to be rounded down to the nearest integer, which we will do by using the method **Math.Floor(...)**.

```
else
{
    Console.WriteLine("It will be a tough winter! More {0} liters wine needed.",
        Math.Floor( ));
}
```

Testing in the Judge System

Test your solution here: <https://judge.softuni.org/Contests/Practice/Index/507#3>.

Problem: Firm

A firm gets a request for creating a project for which a certain number of hours are needed. The firm has a certain number of days. During 10% of the days, the workers are being trained and cannot work on the project. A normal working day is 8 hours long. The project is important for the firm and every worker must work on it with overtime of 2 hours per day.

The hours must be rounded down to the nearest integer (for example, 6.98 hours are rounded to 6 hours).

Write a program that calculates whether **the firm can finish the project on time** and **how many hours more are needed or left**.

Input Data

The input data is read from **the console** and contains **exactly three lines**:

- On **the first** line are **the needed hours** – an integer in the range of [0 ... 200 000].
- On **the second** line are **the days that the firm has** – an integer in the range of [0 ... 20 000].
- On **the third** line are **the number of all workers** – an integer in the range of [0 ... 200].

Output Data

Print **one line** on **the console**:

- If the time is enough:
 - "Yes!{the hours left} hours left."
- If the time is NOT enough:
 - "Not enough time!{additional hours} hours needed."

Sample Input and Output

Input	Output
90	
7	Yes!99 hours left.
3	

Input	Output
99	
3	Not enough time!72 hours needed.
1	

Hints and Guidelines

In order to solve the problem, we will read the input data. Then, we will write a few conditional statements and do some calculations. Finally, we will print the result.

Reading the Input Data

Firstly, we need to decide what **data types** we are going to use for **the input data**.

```
var projectHours = int.Parse(Console.ReadLine());
var availableDays = int.Parse(Console.ReadLine());
var overtimeWorkers = int.Parse(Console.ReadLine());
```

Auxiliary Calculations

The next step is to calculate **the number of total working hours** by multiplying the working days by 8 (every working day is 8 hours long) with the number of workers and then sum them with the overtime. **The working days** equal **90% of the days** that the firm has. **The overtime** equals to the result of the multiplication of the number of workers by 2 (the possible hours of overtime) and then it is multiplied by the number of days that the firm has. From the task requirements we see that **the hours** should be **rounded down to the nearest integer**, which we will do with the method **Math.Floor(...)**.

```
double workDays = availableDays * 0.9;
double overtimeHours = overtimeWorkers * 2 * availableDays;
```

```
double workHours = [REDACTED];
double totalHours = [REDACTED];
```

Checking the Conditions and Printing Output Data

After having done the calculations that are needed to find the value of **the working hours**, now we have to check whether these hours are **enough**, or **some hours are left**.

If **the time is enough**, we print the result that is specified in the task requirements, which in this case is the difference between **the working hours** and **the hours needed** for finishing the project.

If **the time is not enough**, we print the additional hours that are needed for finishing the project. They equal the difference between **the hours for the project** and **the total working hours**.

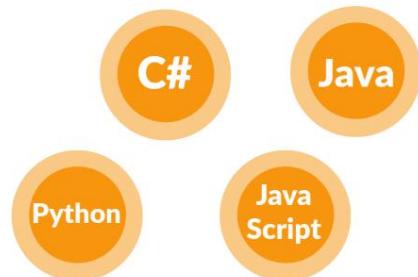
```
if ([REDACTED])
{
    Console.WriteLine("Yes! {0} hours left.",
                      [REDACTED]);
}
else
{
    Console.WriteLine("Not enough time! {0} hours needed.",
                      [REDACTED]);
}
```

Testing in the Judge System

Test your solution here: <https://judge.softuni.org/Contests/Practice/Index/507#4>.

High-quality education profession and job for software engineers

- ✓ Free start for **absolute beginners**
- ✓ Choose among the **most in-demand** professions in the software industry
- ✓ Assistance for **career start**
- ✓ **Top trainers**, mentors and active learning **community**



SoftUni provides practical, modern and **innovative education** for programming, IT, design, marketing and digital skills for thousands of young people. The "**Software University**" program builds true **software engineering professionals**.

Join the end-to-end **software engineering learning program** at SoftUni to master the most in-demand programming concepts, languages, software technologies and skills through a modern teaching methodology, **interactive learning platform** and tons of practical exercises and projects. Our **curriculum** is developed together with the **IT companies** to closely match the industry demands.

SoftUni works directly with the **companies** from the software industry to **find a job** for its students and to make them successful software engineers.

The path of the student at SoftUni



Apply now

softuni.org/apply

Chapter 4.1. More Complex Conditions

In the **current** chapter, we are going to examine the **nested conditional statements** in the **C#** language, by which our program can contain **conditions** that contain other **nested conditional statements**. We call them "nested", because we put an **if** condition into another **if** condition. We are going to examine the **more complex logical conditions** through proper examples.

Video: Chapter Overview

Watch this video to see what you will learn in this chapter: <https://youtu.be/qvbVrKXxsu0>.

Introduction to Complex Conditions by Examples

Conditional statements can be nested, i.e. we can put **if-else** inside another **if-else** statement. Conditions in the **if** constructions can be complex, e.g. use **logical "AND"** or **logical "OR"**. Example:

```
var a = decimal.Parse(Console.ReadLine());
var b = decimal.Parse(Console.ReadLine());

if (a > 0 && b > 0 && a <= 100 && b <= 100)
{
    if (a * b >= 5000)
        Console.WriteLine($"Large size: {a*b}");
    else if (a * b > 1000 && a * b < 5000)
        Console.WriteLine($"Middle size: {a * b}");
    else
        Console.WriteLine($"Small size: {a * b}");
}
else
    Console.WriteLine($"Invalid size (a={a}, b={b})");
```

Run the above code example: <https://repl.it/@nakov/nested-if-else-conditions-csharp>.

The above code performs a **series of checks** using nested **if-else** conditional statements and logical operators like **&&** (logical **AND**) to check the input data for the following 4 cases:

- Size out of range (one of the sides is negative or bigger than 100).
- Large size (area ≥ 5000).
- Middle size ($1000 < \text{area} < 5000$)
- Small size ($\text{area} \leq 1000$)

Let's explain in greater detail how to use **complex and nested if-else conditions** in C#.

Nested If-Else Conditions

Pretty often the program logic requires the use of **if** or **if-else** statements, which are contained one inside another. They are called **nested if** or **if-else** statements. This allows branching the program logic into several levels.

The Nested If-Else Construction

As implied by the definition "nested", these are **if** or **if-else** statements that are placed inside other **if** or **else** statements.

```

if (condition1)
{
    if (condition2)
    {
        // body;
    }
    else
    {
        // body;
    }
}

```

Video: Nested Conditional Statements

Watch a video lesson about the nested if-conditions: <https://youtu.be/4ugMAIkQAMo>.

Deep Nesting

Nesting of **more than three conditional statements** inside each other is not considered a good practice and **has to be avoided**, mostly through optimization of the structure/the algorithm of the code and/or by using another type of conditional statement, which we are going to examine below in this chapter.

Nested If-Else Conditions – Examples

Let's take a few examples in order to gain experience about how to use **nested if-else conditions** in practice.

Example: Personal Titles

Depending on **age** (decimal number) and **gender (m / f)**, print a personal title:

- “Mr.” – a man (gender “m”) – 16 or more years old.
- “Master” – a boy (gender “m”) under 16 years.
- “Ms.” – a woman (gender “f”) – 16 or more years old.
- “Miss” – a girl (gender “f”) under 16 years.

Sample Input and Output

Input	Output	Input	Output	Input	Output	Input	Output
12 f	Miss	17 m	Mr.	25 f	Ms.	13.5 m	Master

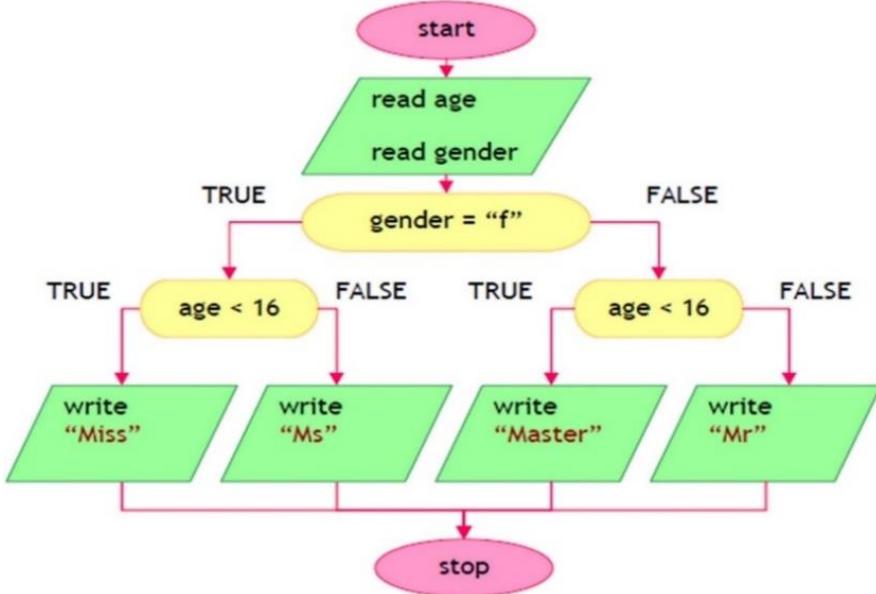
Video: Personal Titles

Watch this video to learn how to solve this problem: <https://youtu.be/7WiBbMOAc7Q>.

Solution

We should notice that the **output** of the program **depends on a few things**. First, we have to check what is the entered **gender** and then check the **age**. Respectively, we are going to use **a few if-else blocks**. These blocks will be **nested**, meaning from **the result** of the first, we are going to **define** which one of the **others** to execute.

The diagram below illustrates the process in detail:



After reading the input data from the console, the following program logic should be executed:

```

var age = double.Parse(Console.ReadLine());
var gender = Console.ReadLine();
if (age < 16)
{
    if (gender == "m") Console.WriteLine("Master");
    else if (gender == "f") Console.WriteLine("Miss");
}
else
{
    if (gender == "m") Console.WriteLine("Mr.");
    else if (gender == "f") Console.WriteLine("Ms.");
}
  
```

Testing in the Judge System

Test your solution here: <https://judge.softuni.org/Contests/Practice/Index/508#0>.

Example: Small Shop

A Bulgarian entrepreneur opens **small shops** in a few cities with different **prices** for the following products:

product / city	Sofia	Plovdiv	Varna
coffee	0.50	0.40	0.45
water	0.80	0.70	0.70
beer	1.20	1.15	1.10
sweets	1.45	1.30	1.35
peanuts	1.60	1.50	1.55

Calculate the price by the given **city** (string), **product** (string) and **quantity** (decimal number).

Sample Input and Output

Input	Output	Input	Output	Input	Output	Input	Output
coffee Varna 2	0.9	peanuts Plovdiv 1	1.5	beer Sofia 6	7.2	water Plovdiv 3	2.1

Video: Small Shop

Watch this video to learn how to solve this problem: https://youtu.be/kU_ru7GK-Mg.

Solution

We convert all of the letters into **lower register** using the function `.ToLower()`, in order to compare products and cities **no matter** what the letters are – small or capital ones.

```
var product = Console.ReadLine().ToLower();
var town = Console.ReadLine().ToLower();
var quantity = double.Parse(Console.ReadLine());
if (town == "sofia")
{
    if (product == "coffee")
        Console.WriteLine(0.50 * quantity);
    // TODO: finish this ...
}
if (town == "varna") // TODO: finish this ...
    if (town == "plovdiv") // TODO: finish this ...
```

Testing in the Judge System

Test your solution here: <https://judge.softuni.org/Contests/Practice/Index/508#1>.

More Complex Conditions

Let's take a look at how we can create more **complex logical conditions** in programming. We can use the logical "AND" (`&&`), logical "OR" (`||`), logical **negation** (`!`) and **brackets** (`()`).

Logical "AND", "OR" and "NOT"

This is a short example that demonstrates the power of logical "AND", logical "OR" and logical "NOT":

```
var animal = "horse";
int speed = 45;

if ((animal == "horse" || animal == "donkey") && (speed > 40))
    Console.WriteLine("Run fast")
else if ((animal == "shark" || animal == "dolphin") && (speed > 45))
    Console.WriteLine("Swim fast")
else if (!(speed > 30 || animal == "turtle"))
    Console.WriteLine("Slow move")
```

We shall explain the logical **AND** (||), the logical **OR** (||), and the logical **NOT** (!) in the next few sections, along with examples and exercises.

The Parenthesis () Operator

Like the rest of the operators in programming, the operators **&&** and **||** have a priority, as in the case **&&** is with higher priority than **||**. The operator **()** serves for **changing the priority of operators** and is being calculated first, just like in mathematics. Using parentheses also gives the code better readability and is considered a good practice.

Example of checking whether a variable belongs to certain ranges:

```
if (x < 0) || ((x >= 5) && (x <= 10)) || (x > 20)
{
    ...
}
```

Logical "AND"

As we saw, in some tasks we have to make **many checks at once**. But what happens when in order to execute some code **more** conditions have to be executed and we **don't want to** make a **negation (else)** for each one of them? The option with nested **if blocks** is valid, but the code would look very **unordered** and for sure – **hard** to read and maintain.

The logical "AND" (operator **&&**) means a few conditions have to be **fulfilled simultaneously**. The table of truthfulness, shown on the right, is applicable.

a	b	a && b
true	true	true
true	false	false
false	true	false
false	false	false

Video: Logical "AND"

Watch this video about the logical "AND" operator in programming: https://youtu.be/V86_z8GWarM.

How the && Operator Works?

The **&&** operator accepts a **couple of Boolean** (conditional) statements, which have a **true** or **false** value, and returns **one** bool statement as a **result**. Using it **instead** of a couple of nested **if blocks**, makes the code **more readable, ordered** and **easy** to maintain. But how does it **work**, when we put a **few** conditions one after another? As we saw above, the logical "AND" returns **true, only** when it accepts as **arguments statements** with value **true**. Respectively, when we have a **sequence** of arguments, the logical "AND" **checks** either until one of the arguments is **over**, or until it **meets** an argument with value **false**.

Example:

```
bool a = true;
bool b = true;
bool c = false;
bool d = true;
bool result = a && b && c && d;
// false (as d is not being checked)
```

The program will run in the **following** way: It **starts** the check from **a**, **reads** it and **accepts** that it has a **true** value, after which it **checks** **b**. After it has **accepted** that **a** and **b** return **true**, it **checks** the

next argument. It gets to **c** and sees that the variable has a **false** value. After the program accepts that the argument **c** has a **false** value, it calculates the expression **before c**, **independent** of what the value of **d** is. That is why the evaluation of **d** is being **skipped** and the whole expression is calculated as **false**.

```
if (x >= x1 && x <= x2 && y >= y1 && y <= y2)
```

Example: Point in a Rectangle

Checks whether point $\{x, y\}$ is placed **inside** the rectangle $\{x_1, y_1\} - \{x_2, y_2\}$. The input data is read from the console and consists of 6 lines: the decimal numbers x_1, y_1, x_2, y_2, x and y (as it is guaranteed that $x_1 < x_2$ and $y_1 < y_2$).

Sample Input and Output

Input	Output	Visualization
2 -3 12 3 8 -1	Inside	<p>A 2D coordinate system with x-axis ranging from 0 to 12 and y-axis ranging from -5 to 5. A rectangle is drawn with vertices labeled x_1, y_1 at (2, -3), x_2, y_2 at (8, 3), and sides at $x=2$, $x=8$, $y=-3$, $y=3$. A point (x, y) is plotted inside the rectangle at approximately (5, -1).</p>

Solution

A point is internal for a given polygon, if the following four conditions are applied **at the same time**:

- The point is placed to the right from the left side of the rectangle.
- The point is placed to the left from the right side of the rectangle.
- The point is placed downwards from the upper side of the rectangle.
- The point is placed upwards from the down side of the rectangle.

```
var x1 = double.Parse(Console.ReadLine());
var y1 = double.Parse(Console.ReadLine());
var x2 = double.Parse(Console.ReadLine());
var y2 = double.Parse(Console.ReadLine());

var x = double.Parse(Console.ReadLine());
var y = double.Parse(Console.ReadLine());

if (x >= x1 && x <= x2 && y >= y1 && y <= y2)
{
    Console.WriteLine("Inside");
}
else
{
    Console.WriteLine("Outside");
}
```

Testing in the Judge System

Test your solution here: <https://judge.softuni.org/Contests/Practice/Index/508#2>.

Logical "OR"

The logical "OR" (operator `||`) means that **at least one** among a few conditions is fulfilled. Similar to the operator `&&`, the logical "OR" accepts a few arguments of `bool` (conditional) type and returns `true` or `false`. We can easily guess that we **obtain** a value `true` every time when at least **one** of the arguments has a `true` value (see the truth table).

a	b	<code>a b</code>
true	true	true
true	false	true
false	true	true
false	false	false

Typical example of the logic of this operator is the following: At school the teacher says: "John or Peter should clean the board". To fulfill this condition (to clean the board), it is possible either just for John to clean it, or just for Peter to clean it, or both of them to do it together.

Video: Logical "OR"

Watch this video about the logical "OR" in programming: <https://youtu.be/e6i-2E66RNU>.

How the `||` Operator Works?

We have already learned what the logical "OR" **represents**. But how is it actually being achieved? Just like with the logical "AND", the program **checks** from left to right **the arguments** that are given. In order to obtain `true` from the expression, it is necessary for **just one** argument to have a `true` value. Respectively, the checking **continues** until an **argument** with **such** value is met or until the arguments **are over**. Here is one **example** of the `||` operator in action:

```
bool a = false;
bool b = true;
bool c = false;
bool d = true;
bool result = a || b || c || d;
// true (as c and d are not being checked)
```

The programs **checks a**, accepts that it has a value `false` and continues. Reaching **b**, it understands that it has a `true` value and the whole **expression** is calculated as `true`, **without** having to check **c** or **d**, because their values **wouldn't change** the result of the expression.

Example: Fruit or Vegetable

Let's check whether a given **product** is a **fruit** or a **vegetable**.

- The "fruits" are **banana**, **apple**, **kiwi**, **cherry**, **lemon** and **grapes**.
- The "vegetables" are **tomato**, **cucumber**, **pepper** and **carrot**.
- Everything else is "**unknown**".

Sample Input and Output

Input	Output
banana	fruit

Input	Output
tomato	vegetable

Input	Output
java	unknown

Solution

We have to use a few conditional statements with logical "OR" (||):

```
var s = Console.ReadLine();
if (s == "banana" || s == "apple" || s == "kiwi" ||
    s == "cherry" || s == "lemon" || s == "grapes")
    Console.WriteLine("fruit");
else if (s == "tomato" || s == "cucumber"
        || s == "pepper" || s == "carrot")
    Console.WriteLine("vegetable");
else
    Console.WriteLine("unknown");
```

Testing in the Judge System

Test your solution here: <https://judge.softuni.org/Contests/Practice/Index/508#3>.

Logical Negation (NOT)

Logical negation (operator !) means a given condition is **not fulfilled** (see the truth table on the right).

The operator ! accepts as an **argument** a bool variable and **returns** its value.

a	!a
true	false
false	true

Video: Logical "NOT"

Watch this video to about the logical "NOT" operator: <https://youtu.be/4U7w2ZSAW4>.

Example: Invalid Number

A given **number** is **valid** if it is in the range [100 ... 200] or it is 0. Do a validation for an **invalid** number.

Sample Input and Output

Input	Output
75	invalid

Input	Output
150	(no output)

Input	Output
220	invalid

Solution

```
var inRange = (num >= 100 && num <= 200) || num == 0;
if (!inRange)
    Console.WriteLine("invalid");
```

Testing in the Judge System

Test your solution here: <https://judge.softuni.org/Contests/Practice/Index/508#4>.

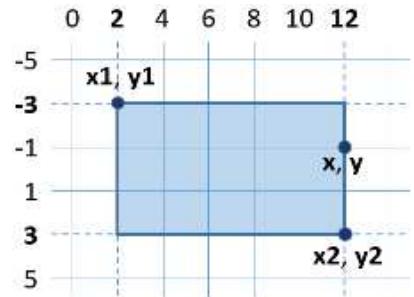
More Complex Conditions – Examples

Sometimes the conditions may be **very complex**, so they can require a long bool expression or a sequence of conditions. Let's take a look at a few examples.

Example: Point on a Rectangle Border

Write a program that checks whether a point $\{x, y\}$ is placed onto any of the sides of a rectangle $\{x_1, y_1\} - \{x_2, y_2\}$ (see the figure).

The input data is read from the console and consists of 6 lines: the decimal numbers x_1, y_1, x_2, y_2, x and y (as it is guaranteed that $x_1 < x_2$ and $y_1 < y_2$). Print "Border" (if the point lies on any of the sides) or "Inside / Outside" (in the opposite case).



Sample Input and Output

Input	Output	Input	Output
2		2	
-3		-3	
12	Border	12	
3		3	Inside / Outside
12		8	
-1		-1	

Solution

The point lies on any of the sides of the rectangle if:

- x coincides with x_1 or x_2 and at the same time y is between y_1 and y_2 or
- y coincides with y_1 or y_2 and at the same time x is between x_1 and x_2 .

This may be checked as follows:

```
if (((x == x1 || x == x2) && (y >= y1) && (y <= y2)) ||
    ((y == y1 || y == y2) && (x >= x1) && (x <= x2)))
{
    Console.WriteLine("Border");
}
```

The previous evaluation might be simplified in the following way:

```
var onLeftSide = (x == x1) && (y >= y1) && (y <= y2);
var onRightSide = (x == x2) && (y >= y1) && (y <= y2);
var onUpSide = (y == y1) && (x >= x1) && (x <= x2);
var onDownSide = (y == y2) && (x >= x1) && (x <= x2);
if (onLeftSide || onRightSide || onUpSide || onDownSide)
{
    Console.WriteLine("Border");
}
```

The second way with the additional Boolean variables is longer, but much more understandable than the first one, isn't it? We recommend when you write logical conditions to make them **easy to read and understand**, instead of making them short. Use additional variables with meaningful names. The names of the variables have to hint what is the value that is kept inside them.

What remains is to finish writing the code to print "Inside / Outside", if the point is not onto any of the sides of the rectangle.

Testing in the Judge System

Test your code here: <https://judge.softuni.org/Contests/Practice/Index/508#5>.

Example: Fruit Shop

A fruit shop sells fruits during **weekdays** and during **weekends** different prices:

Fruit	Weekday price	Weekend price
banana	2.50	2.70
apple	1.20	1.25
orange	0.85	0.90
grapefruit	1.45	1.60
kiwi	2.70	3.00
pineapple	5.50	5.60
grapes	3.85	4.20

Write a program that **reads** from the console a **fruit** (banana / apple / ...), a **day of the week** (Monday / Tuesday / ...) and a **quantity** (a decimal number) and **calculates** the price according to the prices from the tables above. The result has to be printed **rounded up to 2 digits after the decimal point**. Print "error" if it is an **invalid day** of the week or an **invalid name** of a fruit.

Video: Fruit Store

Watch the video to learn how to solve the "Fruit Store" problem: <https://youtu.be/6vZZzil9xBU>.

Sample Input and Output

Input	Output	Input	Output	Input	Output	Input	Output
orange Sunday 3	2.70	kiwi Monday 2.5	6.75	grapes Saturday 0.5	2.10	tomato Monday 0.5	error

Solution

```
if (day == "saturday" || day == "sunday")
{
    if (fruit == "banana") price = 2.70;
    else if (fruit == "apple") price = 1.25;
    // TODO: more fruits come here ...
}
else if (day == "monday" || day == "tuesday" || day ==
    "wednesday" || day == "thursday" || day == "friday")
{
    if (fruit == "banana") price = 2.50;
    // TODO: more fruits come here ...
}
```

Testing in the Judge System

Test your solution here: <https://judge.softuni.org/Contests/Practice/Index/508#6>.

Example: Trade Fees

A company is giving the following **commissions** to its traders according to the **city**, in which they are working and the **volume of sales s**:

City	$0 \leq s \leq 500$	$500 < s \leq 1000$	$1000 < s \leq 10000$	$s > 10000$
Sofia	5%	7%	8%	12%
Varna	4.5%	7.5%	10%	13%
Plovdiv	5.5%	8%	12%	14.5%

Write a **program** that reads the name of a **city** (string) and the volume of **sales** (decimal number) and calculates the rate of the commission fee. The result has to be shown rounded up to 2 digits after the decimal point. When there is an **invalid city or volume of sales** (a negative number), print "error".

Sample Input and Output

Input	Output	Input	Output	Input	Output
Sofia 1500	120.00	Plovdiv 499.99	27.50	Bourgas -50	error

Video: Trade Fees

Watch a video lesson to learn about the "Trade Fees" problem and how to solve it C#: <https://youtu.be/QqKBLJ4JzJ0>.

Solution

When reading the input, we could convert the city into small letters (with the function `.ToLower()`). Initially we set the commission fee to **-1**. It will be changed if the city and the price range are found in the table of commissions. To calculate the commission according to the city and volume of sales, we need a few nested **if statements**, as in the sample code below:

```
var comission = -1.0;
if (town == "sofia")
{
    if (0 <= sales && sales <= 500) comission = 0.05;
    else if (500 < sales && sales <= 1000) comission = 0.07;
    // TODO: check the other price ranges ...
}
else if (town == "varna") // TODO: check the price ranges ...
else if (town == "plovdiv") // TODO: check the price ranges ...
if (comission >= 0)
    Console.WriteLine("{0:f2}", sales * comission);
else Console.WriteLine("error");
```

Testing in the Judge System

Test your solution here: <https://judge.softuni.org/Contests/Practice/Index/508#7>.



It is a good practice to use **blocks** that are **enclosed** with curly braces { } after **if** and **else**. Also, it is recommended during writing to **move aside** the code **after if and else** with a single tabulation **inward**, in order to make the code more easily readable.

Switch-Case Conditional Statement

The **switch-case** condition works as a sequence of **if-else** blocks. Whenever the work of our program depends on the value of **one variable**, instead of making consecutive conditions with **if-else** blocks, we can **use** the conditional **switch** statement. It is being used for **choosing between a list of possibilities**. The statement compares a given value with defined constants and depending on the result, it takes an action.

We put **the variable** that we want to **compare**, inside the **brackets after the operator switch** and it is called a "selector". Here **the type must be comparable** (numbers, strings). **Consecutively**, the program starts **comparing each value** that is **found** after the **case labels**. Upon a match, the execution of the code from the respective place begins and continues until it reaches the operator **break**. In some programming languages (like C and C++) **break** might be skipped, in order to execute a code from other **case** construction, until it reaches another operator. In C# though, the presence of **break** is **mandatory** for **every case** that contains a program logic. When **no matches** are **found**, the **default** construction is being executed, **if** such **exists**.

```
switch (selector)
{
    case value1:
        construction;
        break;
    case value2:
        construction;
        break;
    case value3:
        construction;
        break;
    ...
    default:
        construction;
        break;
}
```

Video: Switch-Case

Watch the video to learn how to use the switch-case conditional statement in programming:
<https://youtu.be/mGJOc4xx5Ho>.

Example: Day of the Week

Let's write a program that prints **the day of the week** (in English) depending on **given number** (1 ... 7) or "**Error!**" if an invalid input is given.

Sample Input and Output

A sample input and output are given in the table on the right.

Input	Output
1	Monday
7	Sunday
-1	Error!

Solution

```
int day = int.Parse(Console.ReadLine());
switch (day)
{
    case 1: Console.WriteLine("Monday"); break;
    case 2: Console.WriteLine("Tuesday"); break;
    ...
    case 7: Console.WriteLine("Sunday"); break;
    default: Console.WriteLine("Error!"); break;
}
```



It is a good practice to put at the first place those **case statements** that process the most common situations and leave the **case constructions** processing the more rear situations at the end, before the **default construction**.

Another good practice is to arrange the **case labels** in ascending order.

Testing in the Judge System

Test your solution here: <https://judge.softuni.org/Contests/Practice/Index/508#8>.

Multiple Labels in Switch-Cases

In C# we have the possibility to use **multiple case** labels in the **switch-case** construction, when they have to execute **the same code**. This way, when our **program** finds a **match**, it will execute the **next** code, because **after** the respective **case** label **there is no code** for execution and a **break** operator.

```
switch (selector)
{
    case value1:
    case value2:
    case value3:
        construction;
        break;
    case value4:
    case value5:
        construction;
        break;
    ...
    default:
        construction;
        break;
}
```

Example: Animal Type

Write a program that prints the type of the animal depending on its name:

- dog -> **mammal**
- crocodile, tortoise, snake -> **reptile**
- others -> **unknown**

Sample Input and Output

Input	Input
tortoise	reptile

Input	Input
dog	mammal

Input	Input
elephant	unknown

Solution

We can solve the task with **switch-case** conditions with multiple labels in the following way:

```
switch (animal)
{
    case "dog": Console.WriteLine("mammal"); break;
    case "crocodile":
    case "tortoise":
    case "snake": Console.WriteLine("reptile"); break;
    default: Console.WriteLine("unknown"); break;
}
```

Testing in the Judge System

Test your solution here: <https://judge.softuni.org/Contests/Practice/Index/508#9>.

Exercises: More Complex Conditions

Now let's **exercise** our new skills with **complex conditions**. Let's solve a few practical tasks.

Video: Chapter Summary

Watch this video to review what we learned in this chapter: <https://youtu.be/QOhyJXZ0HHQ>.

What We Learned in This Chapter?

Before proceeding ahead, let's remind ourselves about the new program constructs and techniques that we have learned in this chapter.

Nested Conditions

```
if (condition1)
{
    if (condition2)
        // body;
    else
        // body;
}
```

Complex Conditions with &&, ||, ! and ()

```
if ((x == left || x == right) && y >= top && y <= bottom)
    Console.WriteLine(...);
```

Switch-Case Statements

```

switch (selector)
{
    case value1:
        construction;
        break;
    case value2:
    case value3:
        construction;
        break;
    ...
    default:
        construction;
        break;
}

```

Problem: Cinema

In a cinema hall the chairs are ordered in a **rectangle** shape in **r** rows and **c** columns. There are three types of screenings with tickets of **different** prices:

- **Premiere** – a premiere screening, with price **12.00** EUR.
- **Normal** – a standard screening, with price **7.50** EUR.
- **Discount** – a screening for children and students on a reduced price – **5.00** EUR.

Write a program that enters a **type of screening** (string), number of **rows** and number of **columns** in the hall (integer numbers) and calculates **the total income** from tickets from a **full hall**. The result has to be printed in the same format as in the examples below – rounded up to 2 digits after the decimal point.

Sample Input and Output

Input	Output
Premiere	
10	
12	1440.00

Input	Output
Normal	
21	
13	2047.50

Hints and Guidelines

While reading the input, we could convert the screening type into small letters (with the function **.ToLower()**). We create and initialize a variable that will store the calculated income. In another variable we calculate the full capacity of the hall. We use a **switch-case** conditional statement to calculate the income according to the type of the projection and print the result on the console in the given format (look for the needed **C#** functionality on the internet). Sample code (parts of the code are blurred with the purpose to stimulate your thinking and problem-solving skills):

```

string type = Console.ReadLine().ToLower();
int rows = [int.Parse(Console.ReadLine())];
int columns = [int.Parse(Console.ReadLine())];
int full = [rows * columns];
double income = [blurred];

```

```

switch (type)
{
    case "premiere":
        income = price * 12.00;
        break;
    case "normal":
        income = price * 7.50;
        break;
    case "discount":
        income = price * 5.00;
        break;
}
Console.WriteLine("Income - " + income);

```

Testing in the Judge System

Test your solution here: <https://judge.softuni.org/Contests/Practice/Index/508#10>.

Problem: Volleyball

Vladimir is a student, lives in Sofia and goes to his hometown from time to time. He is very keen on volleyball but is busy during weekdays and plays **volleyball** only during **weekends** and on **holidays**. Vladimir plays **in Sofia** every **Saturday**, when **he is not working**, and **he is not traveling to his hometown** and also during **2/3 of the holidays**. He travels to his **hometown h times** a year, where he plays volleyball with his old friends on **Sunday**. Vladimir **is not working 3/4 of the weekends**, during which he is in Sofia. Furthermore, during **leap years** Vladimir plays **15% more** volleyball than usual. We accept that the year has exactly **48 weekends**, suitable for volleyball. Write a program that calculates **how many times** Vladimir has played volleyball through the year. Round the result down to the nearest whole number (e.g. 2.15 -> 2; 9.95 -> 9).

The input data is read from the console:

- The first line contains the word “**leap**” (leap year) or “**normal**” (a normal year with 365 days).
- The second line contains the integer **p** – the count of holidays in the year (which are not Saturday or Sunday).
- The third line contains the integer **h** – the count of weekends, in which Vladimir travels to his hometown.

Sample Input and Output

Input	Output
leap	
5	
2	45

Input	Output
normal	
3	
2	38

Input	Output
normal	
11	
6	44

Input	Output
leap	
0	
1	41

Hints and Guidelines

As usual, we read the input data from the console and, to avoid making mistakes, we convert the text into small letters with the function **.ToLower()**. Consequently, we calculate **the weekends spent in Sofia**, **the time for playing in Sofia** and **the common playtime**. At last, we check whether the year is

leap, we make additional calculation when necessary and we print the result on the console **rounded down** to the nearest **integer** (look for a C# class with such functionality).

A sample code (parts of the code are blurred on purpose to stimulate independent thinking and problem-solving skills):

```
string year = Console.ReadLine().ToLower();
int holidays = int.Parse(Console.ReadLine());
int weekendsHome = int.Parse(Console.ReadLine());

int sofiaWeekends = 100 - weekendsHome;
double playSofia = 3.0 * sofiaWeekends / 8 + 3.0 * holidays / 8;
double playTotal = playSofia + weekendsHome;

if (year.Equals("leap"))
{
    playTotal = Math.Floor(playTotal * 35 / 100 + playTotal);
}
else if (year.Equals("normal"))
{
    playTotal = Math.Floor(playTotal);
}

Console.WriteLine(playTotal);
```

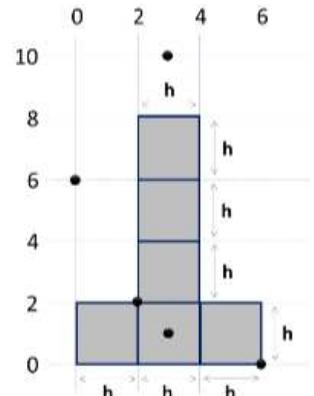
Testing in the Judge System

Test your solution here: <https://judge.softuni.org/Contests/Practice/Index/508#11>.

Problem: * Point in the Figure

The figure consists of 6 blocks with size $h \times h$, placed as in the figure below. The lower left angle of the building is on position {0, 0}. The upper right angle of the figure is on position { $2 \cdot h$, $4 \cdot h$ }. The coordinates given in the figure are for $h = 2$ (see the figure on the right).

Write a program that enters an integer h and the coordinates of a given point $\{x, y\}$ (integers) and prints whether the point is inside the figure (**inside**), outside of the figure (**outside**) or on any of the borders of the figure (**border**).



Sample Input and Output

Input	Output	Input	Output	Input	Output	Input	Output
2 3 10	outside	2 3 1	inside	2 2 2	border	2 6 0	border
Input	Output	Input	Output	Input	Output	Input	Output
2 0 6	outside	15 13 55	outside	15 29 37	inside	15 37 18	outside

Hints and Guidelines

A possible logic for solving the task (not the only correct one):

- We might split the figure into **two rectangles** with a common side:
- A point is **outer (outside)** for the figure, when it is **outside** both of the rectangles.
- A point is **inner (inside)** for the figure, if it is inside one of the rectangles (excluding their borders) or lies on their common side.
- In **other case** the point lies on the border of the rectangle (**border**).

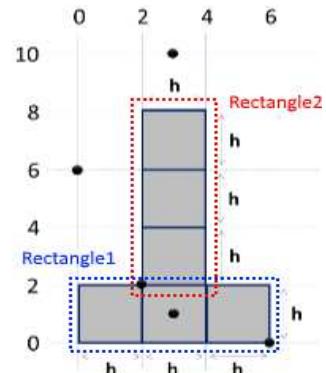
Implementation of the Proposed Idea

An exemplary implementation of the described idea (parts of the code are blurred with the purpose of stimulating logical thinking and solving skills):

```
int h = ...; // Person's height (ReadLine());
int x = ...; // Person's x coordinate (ReadLine());
int y = ...; // Person's y coordinate (ReadLine());

bool outRectangle1 = ...;
bool outRectangle2 = ...;
bool inRectangle1 = ...;
bool inRectangle2 = ...;
bool commonBorder = ...;

if (outRectangle1 || outRectangle2)
{
    Console.WriteLine("outside");
}
else if (inRectangle1 || inRectangle2 || commonBorder)
{
    Console.WriteLine("inside");
}
else
{
    Console.WriteLine("border");
}
```



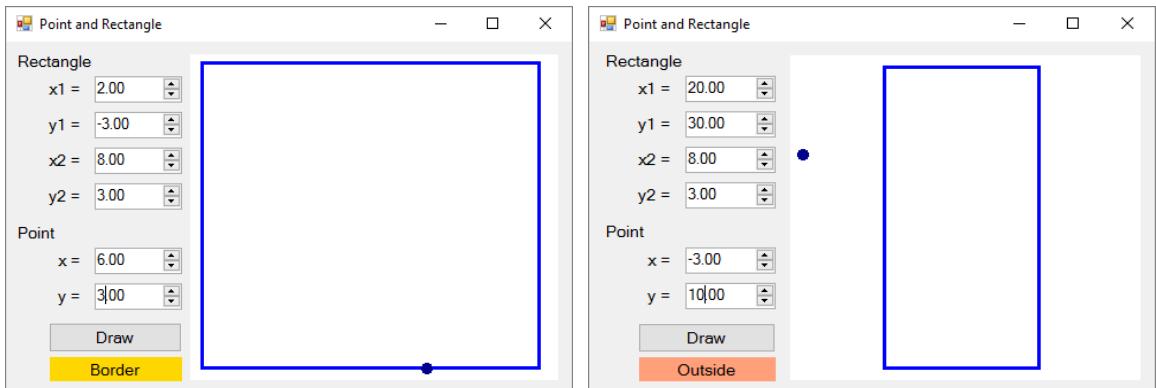
Testing in the Judge System

Test your solution here: <https://judge.softuni.org/Contests/Practice/Index/508#12>.

Lab: * GUI (Desktop) Application: Point and Rectangle

In this chapter we learned how we can use **statements with non-trivial conditions**. Now let's apply this knowledge to create something much more interesting: a **desktop (GUI) application to visualize a point in a rectangle**. This is a wonderful visualization of one of the tasks from the exercises, where a point might lay inside, outside or at the border.

The assignment that we have is to develop a graphical (GUI) app for **visualizing a point and a rectangle**. The application may look like at the screenshots.



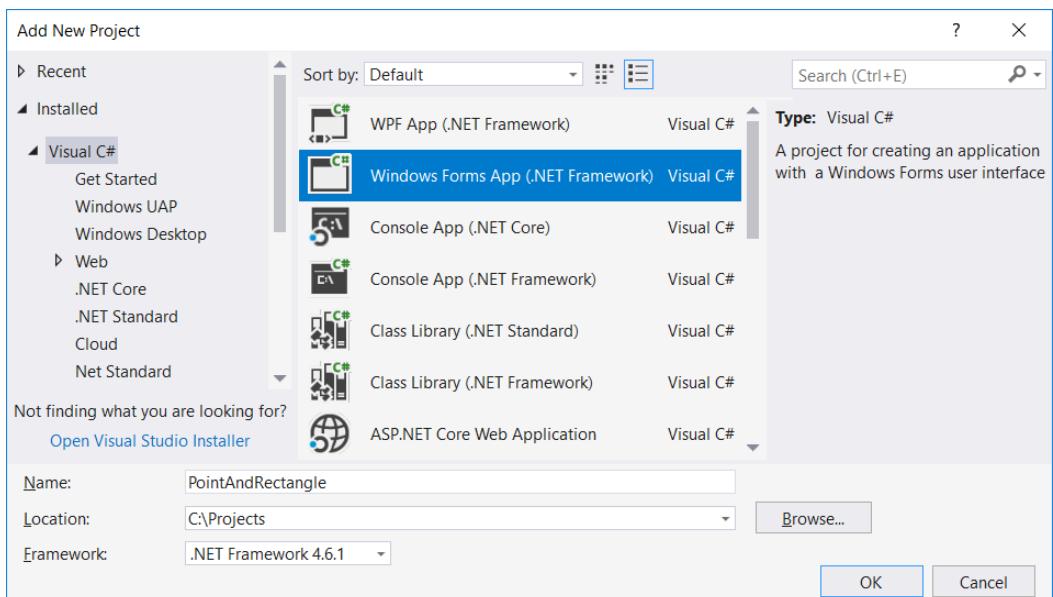
Using the controls on the left we set the coordinates of **two of the angles of the rectangle** (decimal numbers) and the coordinates of the point. The application **visualizes graphically** the rectangle and the point and prints whether the point is **inside** the rectangle (**Inside**), **outside** of it (**Outside**) or on one of its sides (**Border**). The application **moves and resizes** the coordinates of the rectangle and the point to be maximum large, but to fit the field for visualization in the right side of the application.



Attention: this application is significantly **more complex** than the previous graphical applications, which we have developed until now, because it requires using functions for drawing and non-trivial calculations for resizing and moving the rectangle and the point. Instructions for building the application step by step follow.

Creating a New C# Project and Adding Controls

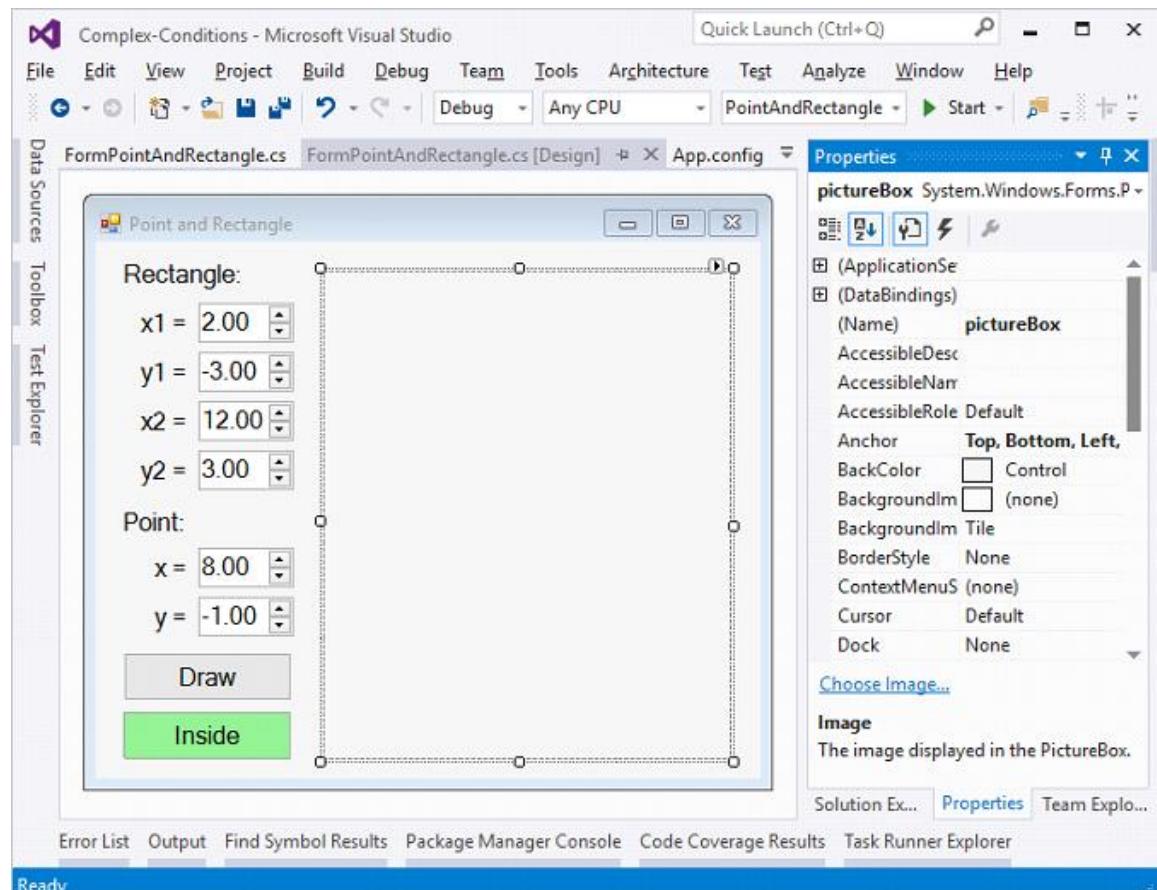
We create a new project of type “**Windows Forms Application**” with a suitable name, for example “**Point-and-Rectangle**”:



We arrange the controls inside the form, as it is shown in the screenshot below:

- 6 boxes for entering a number (**NumericUpDown**), for the **x1**, **y1**, **x2** and **y2** coordinates of the rectangle and for the **x** and **y** coordinates of the point.
- Labels (**Label**) before each box for entering a number.
- A button (**Button**) for drawing the rectangle and the point.
- A text block for the result (**Label**) – the green box at the screenshot.
- A rectangular drawing box (**PictureBox**) for visualizing the rectangle and the point.

We set the **sizes** and **properties** of the controls to look as close as the ones at the screenshot.



Configuring the UI Controls

We set the following recommended settings of the controls:

- For the main form (**Form**) that contains all of the controls:
 - (name) = **FormPointAndRectangle**
 - **Text** = Point and Rectangle
 - **Font.Size** = 12
 - **Size** = 700, 410
 - **MinimumSize** = 500, 400
 - **FormBorderStyle** = FixedSingle

- For the fields for entering a number (`NumericUpDown`):
 - `(name) = numericUpDownX1; numericUpDownY1; numericUpDownX2;`
`numericUpDownY2; numericUpDownX; numericUpDownY`
 - `Value = 2; -3; 12; 3; 8; -1`
 - `Minimum = -100000`
 - `Maximum = 100000`
 - `DecimalPlaces = 2`
- For the button (`Button`) for visualization of the rectangle and the point:
 - `(name) = buttonDraw`
 - `Text = Draw`
- For the text block for the result (`Label`):
 - `(name) = labelLocation`
 - `AutoSize = false`
 - `BackColor = PaleGreen`
 - `TextAlign = MiddleCenter`
- For the field with the draft (`PictureBox`):
 - `(name) = pictureBox`
 - `Anchor = Top, Bottom, Left, Right`

Handling Events

We have to catch the following **events** to write the C# code that will be executed upon their occurrence:

- The event `Click` the button `buttonDraw` (it is called upon pressing the button).
- The event `ValueChanged` of the controls for entering numbers `numericUpDownX1`, `numericUpDownY1`, `numericUpDownX2`, `numericUpDownY2`, `numericUpDownX` and `numericUpDownY` (it is called upon changing the value in the control that enters a number).
- The event `Load` of the form `FormPointAndRectangle` (it is called upon starting the application, before the main form is shown on the display).
- The event `Resize` of the form `FormPointAndRectangle` (it is called upon changing the size of the main form).

All of the above-mentioned events will execute the same action `Draw()`, which will visualize the rectangle and the point and show whether it's inside, outside or onto one of the sides. The code may look like this:

```
private void buttonDraw_Click(object sender, EventArgs e)
{
    Draw();
}

private void FormPointAndRectangle_Load(object sender, EventArgs e)
{
    Draw();
}

private void FormPointAndRectangle_Resize(object sender, EventArgs e)
```

```

{
    Draw();
}

private void numericUpDownX1_ValueChanged(object sender, EventArgs e)
{
    Draw();
}

/* TODO: implement in the same way the event handlers
    numericUpDownY1_ValueChanged,
    numericUpDownX2_ValueChanged,
    numericUpDownY2_ValueChanged,
    numericUpDownX_ValueChanged and
    numericUpDownY_ValueChanged */

private void Draw()
{
    // TODO: implement this a bit later ...
}

```

Printing Point Position Compared to the Rectangle

Let's begin from the easier part: printing the information about the point's position (Inside, Outside or Border). The code must look like this:

```

private void Draw()
{
    // Get the rectangle and point coordinates from the form
    var x1 = this.numericUpDownX1.Value;
    var y1 = this.numericUpDownY1.Value;
    var x2 = this.numericUpDownX2.Value;
    var y2 = this.numericUpDownY2.Value;
    var x = this.numericUpDownX.Value;
    var y = this.numericUpDownY.Value;

    // Display the location of the point: Inside/Border/Outside
    DisplayPointLocation(x1, y1, x2, y2, x, y);
}

private void DisplayPointLocation(decimal x1, decimal y1,
    decimal x2, decimal y2, decimal x, decimal y)
{
    var left = Math.Min(x1, x2);
    var right = Math.Max(x1, x2);
    var top = Math.Min(y1, y2);
    var bottom = Math.Max(y1, y2);
    if (x > left && x < right && ...
    {
        this.labelLocation.Text = "Inside";
        this.labelLocation.BackColor = Color.LightGreen;
    }
    else if (... || y < top || y > bottom)

```

```

    {
        this.labelLocation.Text = "Outside";
        this.labelLocation.BackColor = Color.LightSalmon;
    }
    else
    {
        this.labelLocation.Text = "Border";
        this.labelLocation.BackColor = Color.Gold;
    }
}

```

The code above takes the coordinates of the rectangle and the point and checks whether the point is inside, outside or on the borders of the rectangle. By visualizing the result, the color of the background of the text block that contains it is changed.

Think about how to **finish** the uncompleted (on purpose) conditions in the **if statements!** The code above **purposely doesn't compile**, because the purpose is to make you think about how and why it works and **finish on your own** the missing parts.

Visualization of the Rectangle and the Point

What remains is to implement the most complex part: visualization of the rectangle and the point in the control **pictureBox** with resizing. We can help ourselves with **the code below**, which makes some calculations and draws a blue rectangle and a dark blue circle (the point) according to the coordinates given in the form. Unfortunately, the complexity of the code exceeds the material learned until the present moment and it is complicated to explain in detail exactly how it works. There are comments for orientation. This is the full version of the action **Draw()**:

```

private void Draw()
{
    // Get the rectangle and point coordinates from the form
    var x1 = this.numericUpDownX1.Value;
    var y1 = this.numericUpDownY1.Value;
    var x2 = this.numericUpDownX2.Value;
    var y2 = this.numericUpDownY2.Value;
    var x = this.numericUpDownX.Value;
    var y = this.numericUpDownY.Value;

    // Display the location of the point: Inside/Border/Outside
    DisplayPointLocation(x1, y1, x2, y2, x, y);

    // Calculate the scale factor (ratio) for the
    // diagram holding the rectangle and point in order to
    // fit them well in the picture box
    var minX = Min(x1, x2, x);
    var maxX = Max(x1, x2, x);
    var minY = Min(y1, y2, y);
    var maxY = Max(y1, y2, y);
    var diagramWidth = maxX - minX;
    var diagramHeight = maxY - minY;
    var ratio = 1.0m;
    var offset = 10;
    if (diagramWidth != 0 && diagramHeight != 0)
    {
        var ratioX = (pictureBox.Width - 2 * offset - 1) / diagramWidth;
        var ratioY = (pictureBox.Height - 2 * offset - 1) / diagramHeight;
    }
}

```

```

        ratio = Math.Min(ratioX, ratioY);
    }

    // Calculate the scaled rectangle coordinates
    var rectLeft = offset + (int)Math.Round((Math.Min(x1, x2) - minX) * ratio);
    var rectTop = offset + (int)Math.Round((Math.Min(y1, y2) - minY) * ratio);
    var rectWidth = (int)Math.Round(Math.Abs(x2 - x1) * ratio);
    var rectHeight = (int)Math.Round(Math.Abs(y2 - y1) * ratio);
    var rect = new Rectangle(rectLeft, rectTop, rectWidth, rectHeight);

    // Calculate the scaled point coordinates
    var pointX = (int)Math.Round(offset + (x - minX) * ratio);
    var pointY = (int)Math.Round(offset + (y - minY) * ratio);
    var pointRect = new Rectangle(pointX - 2, pointY - 2, 5, 5);

    // Draw the rectangle and point
    pictureBox.Image = new Bitmap(pictureBox.Width, pictureBox.Height);
    using (var g = Graphics.FromImage(pictureBox.Image))
    {
        // Draw diagram background (white area)
        g.Clear(Color.White);

        // Draw the rectangle (scaled to the picture box size)
        var pen = new Pen(Color.Blue, 3);
        g.DrawRectangle(pen, rect);

        // Draw the point (scaled to the picture box size)
        pen = new Pen(Color.DarkBlue, 5);
        g.DrawEllipse(pen, pointRect);
    }
}

private decimal Min(decimal val1, decimal val2, decimal val3)
{
    return Math.Min(val1, Math.Min(val2, val3));
}

private decimal Max(decimal val1, decimal val2, decimal val3)
{
    return Math.Max(val1, Math.Max(val2, val3));
}

```

In the code above we can see a lot of **conversion of types**, because different types of numbers are used (decimal numbers, real numbers and integers) and sometimes it is required to do conversion between them.

Compiling and Testing the Application

In the end we **compile the code**. If there are errors, we eliminate them. The most probable **reason** for an error is **an inconsistent name of some of the controls** or if **writing the code in the wrong place**. We **start the application** and **test** it. We enter different data to see whether it behaves correctly. If you have problems with the sample project above, you can ask for help in the SoftUni official **discussion forum** (<http://forum.softuni.org>) or in the SoftUni official **Facebook page** (<https://fb.com/softuni.org>).

Chapter 4.2. More Complex Conditions – Exam Problems

The previous chapter introduced you to **nested conditions** in C#. Via nested conditions, the program logic in a particular application can be represented using **if conditional statements** that are nested one into another. We also explained the **switch-case** conditional statement that allows selecting from a list of options.

Now we are going to solve some **exercises** and make sure we have in-depth understanding of the material, by discussing a number of more complex problems that had been given to students on exams.

More Complex Conditions – Quick Review

Before moving to the problems, let's first recall what **nested conditions** are.

Nested Conditions

```
if (condition1)
{
    if (condition2)
        // body;
    else
        // body;
}
```



Remember that it is **not a good practice** to write **deeply nested conditional statements** (with more than three levels of nesting). Avoid nesting of more than three conditional statements inside one another. This complicates the code and makes its reading and understanding difficult.

Switch-Case Conditions

When the program operation depends on the value of a variable, instead of doing consecutive checks with multiple **if-else** blocks, we can use the **switch-case** conditional statement.

```
switch (selector)
{
    case value1:
        statement;
        break;
    case value2:
        statement;
        break;

    default:
        statement;
        break;
}
```

The structure consists of a **selector** (an expression that calculates a particular value) + multiple **case** labels followed by commands, ending in a **break**. The selector type can be an integer, string or

enumeration (**enum**). Now, after we refreshed our knowledge on how to **use and nest conditional statements** in order to implement more complex conditions and program logic, let's solve some **exam** problems.

Problem: On Time for the Exam

A student has to attend **an exam at a particular time** (for example at 9:30 am). They arrive in the exam room at a particular **time of arrival** (for example 9:40 am). It is considered that the student has arrived **on time**, if they have arrived **at the time when the exam starts or up to half an hour earlier**. If the student has arrived **more than 30 minutes earlier**, the student has come **too early**. If they have arrived **after the time when the exam starts**, they are **late**.

Write a program that inputs the exam starting time and the time of student's arrival, and prints if the student has arrived **on time**, if they have arrived **early** or if they are **late**, as well as **how many hours or minutes** the student is early or late.

Sample Input and Output

Input	Output	Input	Output	Input	Output
9 30 9 50	Late 20 minutes after the start	16 00 15 00	Early 1:00 hours before the start	9 00 8 30	On time 30 minutes before the start

Input	Output	Input	Output	Input	Output
9 00 10 30	Late 1:30 hours after the start	14 00 13 55	On time 5 minutes before the start	10 00 10 00	On time

Input Data

Read the following **four integers** (one on each line) from the console:

- The first line contains **exam starting time (hours)** – an integer from 0 to 23.
- The second line contains **exam starting time (minutes)** – an integer from 0 to 59.
- The third line contains **hour of arrival** – an integer from 0 to 23.
- The fourth line contains **minutes of arrival** – an integer from 0 to 59.

Output Data

Print the following on the first line on the console:

- "**Late**", if the student arrives **later** compared to the exam starting time.
- "**On time**", if the student arrives **exactly** at the exam starting time or up to 30 minutes earlier.
- "**Early**", if the student arrives more than 30 minutes **before** the exam starting time.

If the student arrives with more than one minute difference compared to the exam starting time, print on the next line:

- "mm minutes before the start" for arriving less than an hour earlier.
- "hh:mm hours before the start" for arriving 1 hour or earlier. Always print minutes using 2 digits, for example "1:05".
- "mm minutes after the start" for arriving more than an hour late.
- "hh:mm hours after the start" for arriving late with 1 hour or more. Always print minutes using 2 digits, for example "1:03".

Hints and Guidelines

Let's solve the problem step by step.

Processing the Input Data

According to the assignment, we expect **four** lines containing different **integers** to be passed. Examining the provided parameters, we can use the **int** type, as it is suitable for the expected values. We simultaneously **read** the input data and **parse** the string value to the selected data type for **integer**.

```
var examHours = int.Parse(Console.ReadLine());
var examMinutes = int.Parse(Console.ReadLine());
var arrivalHours = int.Parse(Console.ReadLine());
var arrivalMinutes = int.Parse(Console.ReadLine());
```

Examining the expected output, we can create variables that contain the different output data types, in order to avoid using the so called "**magic strings**" in the code.

```
string late = "Late";
string onTime = "On time";
string early = "Early";
```

Calculating Exam Start Time and Student Arrival Time

After reading the input data, we can now start writing the logic for calculating the result. Let's first calculate the **start time** of the exam **in minutes** for easier and more accurate comparison.

```
int examTime = (examHours * 60) + examMinutes;
```

Let's also calculate the **student arrival time** using the same logic.

```
int arrivalTime =
    (arrivalHours * 60) + arrivalMinutes;
```

What remains is to calculate the difference between the two times, in order to determine **when** and **what time compared to the exam time** the student arrived at.

```
int totalMinutesDifference = arrivalTime - examTime;
```

Checking If the Student Arrived on Time or Late

Our next step is to do the required **checks and calculations**, and finally we will print the output. Let's separate the code into **two** parts:

- First, let's show when the student arrived – were they **early**, **late** or **on time**. In order to do that, we will use an **if-else** statement.
- After that, we will show the **time difference**, if the student arrives in a **different time** compared to the **exam starting time**.

In order to spare one additional check (**else**), we can, by default, assume that the student was late.

After that, according to the condition, we will check whether the difference in times is **more than 30 minutes**. If this is true, we assume that the student is **early**. If we do not match the first condition, we need to check if **the difference is less than or equal to zero (≤ 0)**, by which we are checking the condition whether the student arrived within the range of **0 to 30 minutes** before the exam.

In all other cases we assume that the student **was late**, which we set as **default**, and no additional check is needed.

```
string studentArrival = late;
if (totalMinutesDifference < -30)
{
    studentArrival = early;
}
else if (totalMinutesDifference <= 30)
{
    studentArrival = onTime;
}
```

Calculating Time Difference

Finally, we need to understand and print **what is the time difference between exam start time and student arrival time**, as well as whether this time difference indicates time of arrival **before or after** the exam start.

We check whether the time difference is **more than** one hour, in order to print hours and minutes in the required **format**, or **less than** one hour, in order to print **only minutes** as a format and description.

We need to do one more check: whether the time of student's arrival is **before** or **after** the exam start.

```
string result = string.Empty;
if (totalMinutesDifference != 0)
{
    int hoursDifference =
        Math.Abs(totalMinutesDifference / 60);
    int minutesDifference =
        Math.Abs(totalMinutesDifference % 60);

    if (hoursDifference > 0)
    {
        result = string.Format("{0}:{1:00} hours",
                               hoursDifference, minutesDifference);
    }
    else
    {
        result = minutesDifference + " minutes";
    }

    if (totalMinutesDifference < 0)
    {
        result += " before the start";
    }
}
```

```

    else
    {
        result += " after the start";
    }
}

```

Printing the Result

Finally, what remains is to print the result on the console. According to the requirements, if the student arrived right on time (**not even a minute difference**), we do not need to print a second result. This is why we apply the following condition:

```

Console.WriteLine(studentArrival);
if (!string.IsNullOrEmpty(result))
{
    Console.WriteLine(result);
}

```

Actually, for the purposes of the task, printing the result **on the console** can be done on a much earlier stage – during the calculations. This, however, is not a very good practice. **Why?**

Let's examine the idea that our code is not 10 lines, but 100 or 1000! One day, printing the result will not be done on the console, but will be written in a **file** or displayed as a **web application**. Then, how many places in the code you will make changes at, due to such a correction? Are you sure you won't miss some places?



Always consider the code that contains logical calculations as a separate part, different from the part that processes the input and output data. It has to be able to work regardless of how the data is passed to it and where the result will be displayed.

Testing in the Judge System

Test your solution here: <https://judge.softuni.org/Contests/Practice/Index/509#0>.

Problem: Trip

It is strange, but most people start planning their vacations well in advance. A young programmer from Bulgaria has **certain budget** and spare time in a particular **season**.

Write a program that accepts **as input the budget** (in BGN – Bulgarian levs) and **season**, and **as output** displays programmer's **vacation place** and **the amount of money they will spend**.

The **budget determines the destination**, and the **season determines what amount of the budget will be spent**. If the season is **summer**, the programmer will go **camping**, if it is **winter** – they will stay in a **hotel**. If it is in **Europe**, **regardless of the season**, the programmer will stay in a **hotel**. Each **camp** or **hotel**, according to the **destination**, has its own **price**, which corresponds to a particular **percentage of the budget**:

- If **100 BGN or less** – somewhere in **Bulgaria**.
 - **Summer** – **30%** of the budget.
 - **Winter** – **70%** of the budget.
- If **1000 BGN or less** – somewhere on the **Balkans**.
 - **Summer** – **40%** of the budget.

- Winter – 80% of the budget.
- If more than 1000 BGN – somewhere in Europe.
 - Upon traveling in Europe, regardless of the season, the programmer will spend 90% of the budget.

Input Data

The input data will be read from the console and will consist of **two lines**:

- The **first** line holds **the budget** – **real number** in the range [10.00 ... 5000.00].
- The **second** line holds **one** of two possible seasons: "summer" or "winter".

Output Data

Two lines must be printed on the console.

- On the **first** line – "Somewhere in {destination}" among "Bulgaria", "Balkans" and "Europe".
- On the **second** line – "{Vacation type} – {Amount spent}".
 - The **Vacation** can be in a "Camp" or "Hotel".
 - The **Amount** must be rounded up to the second digit after the decimal point.

Sample Input and Output

Input	Output
50 summer	Somewhere in Bulgaria Camp – 15.00

Input	Output
75 winter	Somewhere in Bulgaria Hotel – 52.50

Input	Output
312 summer	Somewhere in Balkans Camp – 124.80

Input	Output
1500 summer	Somewhere in Europe Hotel – 1350.00

Hints and Guidelines

Typically, as for the other tasks, we can separate the solution into the following parts: reading the input data, doing calculations, printing the result.

Processing the Input Data

While reading carefully the requirements, we understand that we expect **two** lines of input data. The first parameter is a **real number**, for which we need to pick an appropriate variable type. For higher level of calculation accuracy, we can pick **decimal** as a variable for the budget and – **string** for the season.

```
decimal budget = decimal.Parse(Console.ReadLine());
string season = Console.ReadLine();
```



Always take into consideration what **value type** is passed in the input data, as well as what type these need to be converted to, in order for the program conditions to work properly!

Example: When you need to do money calculations in a task, use **decimal** for higher level of accuracy.

Calculations

Let's create and initialize the variables needed for applying the logic and calculations.

```
string destinationResult = string.Empty;
string holidayInformation = string.Empty;
decimal moneySpent = 0.00M;
```

Similarly to the example in the previous task, we can initialize variables with some of the output results, in order to spare additional initialization.

When examining once again the problem requirements, we notice that the main distribution of where the vacation will take place is determined by the **value of the budget**, i.e. our main logic is divided into two cases:

- If the budget is **less than** a particular value.
- If it is **less than** another value or is **more than** the specified border value.

Based on the way we arrange the logical scheme (the order in which we will check the border values), we will have more or less conditions in the solution. **Why?**

After that, we need to apply a condition to check the value of the **season**. Based on it, we will determine what percentage of the budget will be spent, as well as where the programmer will stay – in a **hotel** or a **camp**.

This is a sample code that may be used to implement the above idea:

```
if (budget <= 100.00M)
{
    destinationResult = "Bulgaria";
    if (season.Equals("summer"))
    {
        moneySpent = 0.30M * budget;
        holidayInformation =
            string.Format("Camp - {0:F2}", moneySpent);
    }
    else
    {
        moneySpent = 0.70M * budget;
        holidayInformation =
            string.Format("Hotel - {0:F2}", moneySpent);
    }
}
else if (budget <= 1000.00M)
{
    destinationResult = "Balkans";
    if (season.Equals("summer"))
    {
        moneySpent = 0.40M * budget;
```

```

        holidayInformation =
            string.Format("Camp - {0:F2}", moneySpent);
    }
    else
    {
        moneySpent = 0.80M * budget;
        holidayInformation =
            string.Format("Hotel - {0:F2}", moneySpent);
    }
}

else
{
    destinationResult = "Europe";
    moneySpent = 0.90M * budget;
    holidayInformation =
        string.Format("Hotel - {0:F2}", moneySpent);
}

```

Printing the Result

What remains is to display the calculated result on the console:

```

Console.WriteLine("Somewhere in {0}", destinationResult);
Console.WriteLine(holidayInformation);

```

Testing in the Judge System

Test your solution here: <https://judge.softuni.org/Contests/Practice/Index/509#1>.

Problem: Operations with Numbers

Write a program that reads **two integers** (**n1** and **n2**) and an **operator** that performs a particular **mathematical operation** with them. Possible operations are: **summing up** (+), **subtraction** (-), **multiplying** (*), **division** (/) and **modular division** (%). Upon summing up, subtracting and multiplying, the console must print the result and display whether it is **even** or **odd** number. Upon regular division – just the **result**, and upon modular division – the **remainder**. You need to take into consideration the fact that the divisor can be equal to zero (= 0) and dividing by zero is not possible. In this case, a **special notification** must be printed.

Input Data

3 lines are read from the console:

- N1 – integer within the range [0 ... 40 000].
- N2 – integer within the range [0 ... 40 000].
- Operator – one character among: "+", "-", "*", "/", "%".

Output Data

Print the output as a **single line** on the console:

- If the operation is **summing up**, **subtraction** or **multiplying**:
 - " $\{N1\} \{operator\} \{N2\} = \{output\}$ " - {even/odd}".
- If the operation is **division**:
 - " $\{N1\} / \{N2\} = \{output\}$ " – the result is **formatted up to the second digit after the decimal point**.
- If the operation is **modular division**:
 - " $\{N1\} \% \{N2\} = \{remainder\}$ ".
- In case of **dividing by 0 (zero)**:
 - "Cannot divide $\{N1\}$ by zero".

Sample Input and Output

Input	Output	Input	Output	Input	Output
10 1 -	10 - 1 = 9 - odd	7 3 *	7 * 3 = 21 - odd	10 12 +	10 + 12 = 22 - even
7 3 *	7 * 3 = 21 - odd	123 12 /	123 / 12 = 10.25	10 3 %	10 % 3 = 1

Hints and Guidelines

The problem is not complex, but there are a lot of code lines to write.

Processing the Input Data

Upon reading the requirements, we understand that we expect **three** lines of input data. The first **two** lines enter two **integers** (within the specified range), and the third line – **an arithmetical symbol**.

```
decimal n1 = decimal.Parse(Console.ReadLine());
decimal n2 = decimal.Parse(Console.ReadLine());
string nOperator = Console.ReadLine();
```

Condition for 0

Let's create and initialize the variables needed for the logic and calculations. In one variable we will store **the calculations output**, and the other one we will use for the **final output** of the program.

```
decimal result = 0.00M;
string output = string.Empty;
```

When carefully reading the requirements, we understand that there are cases where we don't need to do **any** calculations, and simply display a result.

Therefore, we can first check if the second number is **0** (zero), as well as whether the operation is **division** or **modular division**, and then initialize the output.

```
if (n2 == 0 && (nOperator.Equals("/") || nOperator.Equals("%")))
{
    output = string.Format("Cannot divide {0} by zero", n1);
}
```

Let's place the output as a value upon initializing the **output** parameter. This way we can apply **only one condition** – whether it is needed to **recalculate** or **replace** this output.

Based on the approach that we choose, our next condition will be either a simple **else** or a single **if**. In the body of this condition, using additional conditions regarding the manner of calculating the output based on the passed operator, we can separate the logic based on the **structure** of the expected **output**.

Condition for Division and Modular Division

From the requirements we can see that for **summing up** (+), **subtraction** (-) or **multiplying** (*) the expected output has the same structure: "**{n1} {operator} {n2} = {output}** – {even/odd}", whereas for **division** (/) and **modular division** (%) the output has a different structure.

```
else if (nOperator.Equals("/"))
{
    result = n1 / n2;
    output = string.Format("{0} {1} {2} = {3:F2}",
                           n1, nOperator, n2, result);
}
else if (nOperator.Equals("%"))
{
    result = n1 % n2;
    output = string.Format("{0} {1} {2} = {3}",
                           n1, nOperator, n2, result);
}
```

Condition for Sum, Subtract and Multiply

We finish the solution by applying conditions for summing up, subtraction and multiplying:

```
else
{
    if (nOperator.Equals("+"))
    {
        result = n1 + n2;
    }
    else if (nOperator.Equals("-"))
    {
        result = n1 - n2;
    }
    else if (nOperator.Equals("*"))
    {
        result = n1 * n2;
    }
}
```

```

        output = string.Format("{0} {1} {2} = {3} - {4}",
            n1, nOperator, n2, result,
            result % 2 == 0 ? "even" : "odd");
    }
}

```

For short and clear conditions, such as the above example for even and odd number, you can use a **ternary operator**. Let's examine the possibility to apply a condition **with** or **without** a ternary operator.

Using Ternary Operator

Without using a **ternary operator**, the code is longer but easier to read:

```

string numberIs = string.Empty;
if (result % 2 == 0)
{
    numberIs = "even";
}
else
{
    numberIs = "odd";
}

```

Upon using a **ternary operator**, the code is much shorter but may require additional efforts to read and understand the logic:

```
string numberIs = result % 2 == 0 ? "even" : "odd";
```

Printing the Output

Finally, what remains is to print the calculated result on the console:

```
Console.WriteLine(output);
```

Testing in the Judge System

Test your solution here: <https://judge.softuni.org/Contests/Practice/Index/509#2>.

Problem: Game Tickets

A group of football fans decided to buy tickets for Euro Cup 2016. The tickets are sold in Bulgarian leva (BGN) in **two** price categories:

- VIP – 499.99 BGN (Bulgarian leva)
- Normal – 249.99 BGN (Bulgarian leva)

The football fans **have a shared budget**, and the **number of people** in the group determines what percentage of the budget will be **spent on transportation**:

- 1 to 4 – 75% of the budget
- 5 to 9 – 60% of the budget
- 10 to 24 – 50% of the budget
- 25 to 49 – 40% of the budget
- 50 or more – 25% of the budget

Write a program that calculates whether the money left in the budget will be enough for the football fans to buy tickets in the selected category, as well as how much money they will have left or be insufficient.

Input Data

The input data is read from the **console** and contains **exactly 3 lines**:

- The **first** line holds the **budget** – real number within the range [1 000.00 ... 1 000 000.00].
- The **second** line holds the **category** – "VIP" or "Normal".
- The **third** line holds the **number of people in the group** – an integer within the range [1 ... 200].

Output Data

Print the following on the console as **one line**:

- If the **budget** is sufficient:
 - "Yes! You have {N} leva left." – where **N** is the amount of remaining money for the group.
- If the **budget** is NOT sufficient:
 - "Not enough money! You need {M} leva." – where **M** is the amount that is insufficient.

The amounts must be formatted up to **the second digit after the decimal point**.

Sample Input and Output

Input	Output	Explanations
1000 Normal 1	Yes! You have 0.01 leva left.	1 person: 75% of the budget are spent on transportation. Remaining amount: $1000 - 750 = 250$. Category Normal: the ticket price is $249.99 * 1 = 249.99$ $249.99 < 250$: the person will have $250 - 249.99 = 0.01$ money left

Input	Output	Explanations
30000 VIP 49	Not enough money! You need 6499.51 leva.	49 persons: 40% of the budget are spent on transportation. Remaining amount: $30000 - 12000 = 18000$. Category VIP: the ticket costs $499.99 * 49$. $24499.51 < 18000$: the amount is not enough, more money needed: $24499.51 - 18000 = 6499.51$

Hints and Guidelines

We will read the input data and perform the calculations described in the task requirements, in order to check if the money will be sufficient.

Processing the Input Data

Let's read carefully the requirements and examine what we expect to take as **input data**, what is expected to **return as a result**, as well as what are the **main steps** for solving the problem.

For a start, let's process and save the input data in **appropriate variables**:

```
decimal budget = decimal.Parse(Console.ReadLine());
string ticketType = Console.ReadLine();
int people = int.Parse(Console.ReadLine());
```

Calculating Transportation Costs

Let's create and initialize the variables needed for doing the calculations:

```
decimal transportCharges = 0.00M;
decimal moneyForTickets = 0.00M;
decimal moneyDifference = 0.00M;
```

Let's review the requirements once again. We need to perform **two** different block calculations.

By the first set of calculations we must understand what part of the budget has to be spent on **transportation**. You will notice that the logic for doing these calculations only depends on the **number of people in the group**. Therefore, we will do a logical breakdown according to the number of football fans.

We will use conditional statement – a sequence of **if-else** blocks.

```
if (people <= 4)
{
    transportCharges = 0.75M * budget;
}
else if (people <= 9)
{
    transportCharges = 0.60M * budget;
}
else if (people <= 24)
{
    transportCharges = 0.50M * budget;
}

else if (people <= 49)
{
    transportCharges = 0.40M * budget;
}
else if (people >= 50)
{
    transportCharges = 0.25M * budget;
}
```

Calculating Ticket Costs

By the second set of calculations we need to find out what amount will be needed for purchasing **tickets for the group**. According to the requirements, this only depends on the type of tickets that we need to buy. Let's use a **switch-case** conditional statement:

```
switch (ticketType)
{
    case "Normal":
        moneyForTickets = people * 249.99M;
        break;
    case "VIP":
        moneyForTickets = people * 499.99M;
        break;
    default:
        moneyForTickets = people * 249.99M;
        break;
}
```

Calculating Total Costs

Once we have calculated the **transportation costs** and **ticket costs**, what remains is to calculate the end result and understand if the group of football fans will **attend** Euro Cup 2016 or **not**, provided the available parameters.

For the output, in order to spare one **else condition** in the construction, we will assume that the group can, by default, attend Euro Cup 2016.

```
moneyDifference =
    budget - transportCharges - moneyForTickets;
string result =
    string.Format("Yes! You have {0:F2} leva left.",
        decimal.Round(moneyDifference, 2));
if (moneyDifference < 0)
{
    result = string.Format(
        "Not enough money! You need {0:F2} leva.",
        Math.Abs(decimal.Round(moneyDifference, 2)));
}
```

Printing the Result

Finally, we need to display the calculated result on the console.

Testing in the Judge System

Test your solution here: <https://judge.softuni.org/Contests/Practice/Index/509#3>.

Problem: Hotel Room

A hotel offers **two types of rooms: studio and apartment**. Prices are in Bulgarian levs (BGN / lv).

Write a program that calculates **the price of the whole stay for a studio and apartment**. Prices depend on the **month** of the stay:

May and October	June and September	July and August
Studio – 50 BGN/night	Studio – 75.20 BGN/night	Studio – 76 BGN/night
Apartment – 65 BGN/night	Apartment – 68.70 BGN/night	Apartment – 77 BGN/night

The following **discounts** are also offered:

- For a **studio**, in case of **more than 7** stays in **May and October**: **5%** discount.
- For a **studio**, in case of **more than 14** stays in **May and October**: **30%** discount.
- For a **studio**, in case of **more than 14** stays in **June and September**: **20%** discount.
- For an **apartment**, in case of **more than 14** stays, no limitation regarding the month: **10%** discount.

Input Data

The input data is read from the **console** and contains **exactly two lines**:

- The **first** line contains the **month** – May, June, July, August, September or October.
- The **second** line is the **number of stays** – integer within the range [0 ... 200].

Output Data

Print the following **two lines** on the console:

- On the **first** line: "Apartment: { price for the whole stay } lv."
- On the **second** line: "Studio: { price for the whole stay } lv."

The price for the whole stay must be formatted up to two symbols after the decimal point.

Sample Input and Output

Input	Output	Comments
May 15	Apartment: 877.50 lv. Studio: 525.00 lv.	In May , in case of more than 14 stays , the discount for a studio is 30% ($50 - 15 = 35$), and for the apartment is 10% ($65 - 6.5 = 58.5$). The whole stay in the apartment : 877.50 lv. The whole stay in the studio : 525.00 lv.

Input	Output
June 14	Apartment: 961.80 lv. Studio: 1052.80 lv

Input	Output
August 20	Apartment: 1386.00 lv. Studio: 1520.00 lv.

Hints and Guidelines

We will read the input data and do the calculations according to the provided price list and the discount rules, and finally print the result.

Processing the Input Data

According to the task requirements we expect to read two lines of input data: **the month in which the stay is planned** (first line), and **the number of stays** (second line).

Let's process and store the input data in the appropriate parameters:

```
string month = Console.ReadLine();
int nights = int.Parse(Console.ReadLine());
```

Creating Helper Variables

Now let's create and initialize the variables needed for the calculations:

```
decimal studioPrice = 50.00M;
decimal apartmentPrice = 65.00M;
decimal studioRent = 0.00M;
decimal apartmentRent = 0.00M;
```

When doing an additional analysis of the requirements, we understand that our main logic depends on what **month** is passed and what is the number of **stays**.

In general, there are different approaches and ways to apply the above conditions, but let's examine a basic **switch-case** conditional statement, as in the individual **case blocks** we will use **if** and **if-else** conditional statements.

Calculating Prices for May and October

Let's start with the first group of months: **May** and **October**. For these two months **the price for stay is the same** for both types of accommodation – in a **studio** or in an **apartment**. Therefore, the only thing that remains is to apply an internal condition regarding the **number of stays** and recalculate **the relevant price** (if needed).

```
switch (month)
{
    case "May":
    case "October":
        studioPrice = 50.00M;
        apartmentPrice = 65.00M;

        studioRent = studioPrice * nights;
        apartmentRent = apartmentPrice * nights;

        if (nights > 14)
        {
            studioRent *= 0.70M;
            apartmentRent *= 0.90M;
        }
        else if (nights > 7)
        {
            studioRent *= 0.95M;
        }

        break;
}
```

Calculating Prices for June, September, July and August

To some extent, the **logic** and **calculations** will be **identical** for the following months.

```
case "June":
case "September":
    studioPrice = 75.20M;
    apartmentPrice = 68.70M;

    studioRent = studioPrice * nights;
    apartmentRent = apartmentPrice * nights;

    if (nights > 14)
    {
        studioRent *= 0.80M;
        apartmentRent *= 0.90M;
    }
```

```

break;

case "July":
case "August":
    studioPrice = 76.00M;
    apartmentPrice = 77.00M;

    studioRent = studioPrice * nights;
    apartmentRent = apartmentPrice * nights;

    if (nights > 14)
    {
        apartmentRent *= 0.90M;
    }

break;

```

Formatting the Output Data

After calculating the relevant prices and the total amount for the stay, now let's prepare the formatted result. Before that, we should store it in our output parameters – `studioInfo` and `apartmentInfo`.

```

string studioInfo =
    string.Format("Studio: {0:F2} lv.",
        decimal.Round(studioRent, 2));

string apartmentInfo =
    string.Format("Apartment: {0:F2} lv.",
        decimal.Round(apartmentRent, 2));

```

In order to calculate the output parameters, we will use the `decimal.Round(Decimal, Int32)` method. This method rounds the decimal number up to a specified number of characters after the decimal point. In our case, we will round the decimal number up to 2 digits after the decimal point.

Printing the Result

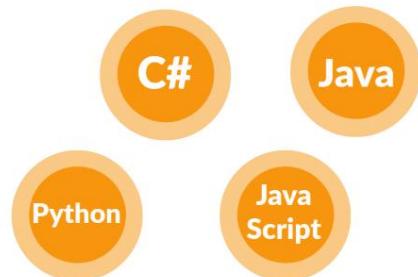
Finally, what remains is to print the calculated results on the console.

Testing in the Judge System

Test your solution here: <https://judge.softuni.org/Contests/Practice/Index/509#4>.

High-quality education profession and job for software engineers

- ✓ Free start for **absolute beginners**
- ✓ Choose among the **most in-demand** professions in the software industry
- ✓ Assistance for **career start**
- ✓ **Top trainers**, mentors and active learning **community**



SoftUni provides practical, modern and **innovative education** for programming, IT, design, marketing and digital skills for thousands of young people. The "**Software University**" program builds true **software engineering professionals**.

Join the end-to-end **software engineering learning program** at SoftUni to master the most in-demand programming concepts, languages, software technologies and skills through a modern teaching methodology, **interactive learning platform** and tons of practical exercises and projects. Our **curriculum** is developed together with the **IT companies** to closely match the industry demands.

SoftUni works directly with the **companies** from the software industry to **find a job** for its students and to make them successful software engineers.

The path of the student at SoftUni



Apply now

softuni.org/apply

Chapter 5.1. Loops (Repetitions)

In the present chapter we will get familiar how to **repeat blocks of commands**, also known in software development as "**loops**". We will write a number of **simple loops** using the **for** operator in its simplest form (**for i = 1 ... n**). Finally, we will solve series of practical problems that require repeating of series of actions, using loops.

Video: Chapter Overview

Watch a video lesson to review what will we learn in this chapter about loops in programming:
<https://youtu.be/GIE2smfXg2g>.

Introduction to Simple Loops by Examples

In programming we can **execute a block of code multiple times** using a simple **for-loop** like this:

```
for (int i = 1; i <= 5; i++)
{
    Console.WriteLine(i);
}
```

Run the above code example: <https://repl.it/@nakov/for-loop-1-to-5-csharp>.

The above code prints the numbers 1, 2, ..., 5. The output is as follows:

```
1
2
3
4
5
```

We can **enter multiple numbers from the console** and process them using loops like this:

1. Read the **count n** of the numbers.
2. In a **for-loop** read and process **n** times one single number.

This is how the above idea may work:

```
int n = int.Parse(Console.ReadLine());
long sum = 0;
for (int i = 0; i < n; i++)
{
    int num = int.Parse(Console.ReadLine());
    sum += num;
}
Console.WriteLine("Sum = {0}", sum);
```

Run the above code example: <https://repl.it/@nakov/for-loop-sum-n-numbers-csharp>.

The **output** from the above example may look like this (when we enter 3 numbers: **10**, **20** and **30**):

```
3
10
20
30
Sum = 60
```

Let's explain in greater detail how to use **simple for loops** to repeat blocks of code multiple times.

For Loops (Repeating Code Blocks)

In programming it is often required to perform a block of commands multiple times. In order to do that, the so-called **loops** are used. Let's examine an example of a **for loop** that passes sequentially through the numbers from 1 to 10 and prints them:

```
for (int i = 1; i <= 10; i++)
{
    Console.WriteLine("i = " + i);
}
```

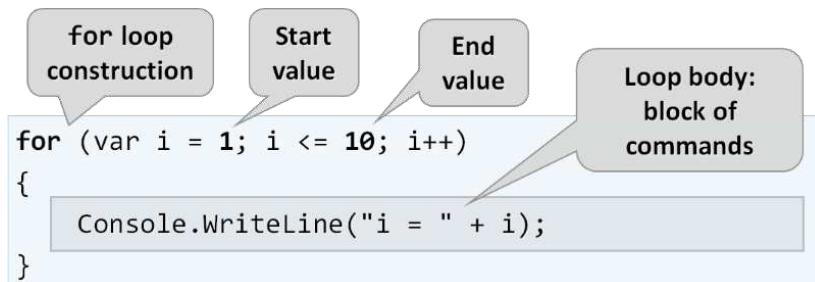
The loop starts with the **for operator** and passes through all values for a particular variable in a given range, for example the numbers from 1 to 10 (included), and for each value it performs a series of commands.

Video: Simple For-Loops

Watch the video about the for-loop statement: <https://youtu.be/yzEamf5L1ZY>.

Syntax: For-Loop

Upon declaring the loop, you can specify a **start value** and an **end value**. The **body of the loop** is usually enclosed in curly brackets {} and represents a block of one or multiple commands. The figure below shows the structure of a **for loop**:



In most cases a **for loop** is run between **1** and **n** times (for example from 1 to 10). The purpose of the loop is to pass sequentially through the numbers 1, 2, 3, ..., n and for each of them to perform a particular action. In the example above, the **i** variable accepts values from 1 to 10 and the current value is printed in the body of the loop. The loop repeats 10 times and each of these repetitions is called an "**iteration**".

Now, let's demonstrate how to use the for loop in practice by a few simple **examples**.

Example: Numbers from 1 to 100

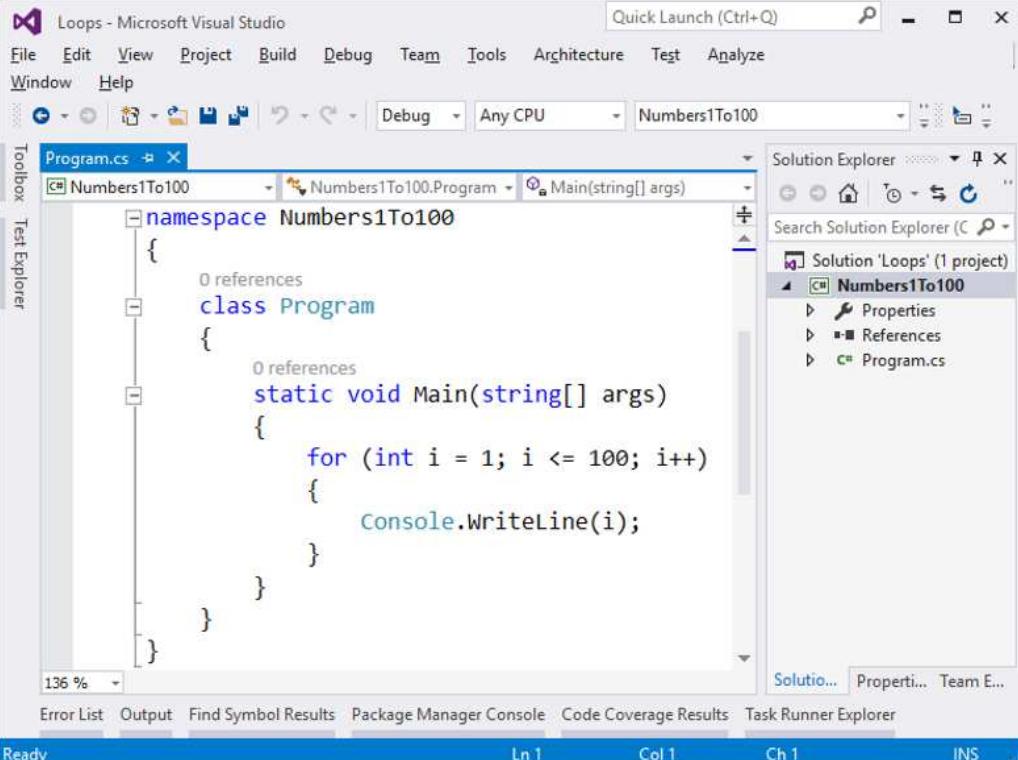
Write a program that **prints the numbers from 1 to 100**. The program does not accept input and prints the numbers from 1 to 100 sequentially, each on a separate line.

Video: Numbers 1...100

Watch the video lesson to learn how to print the numbers from 1 to 100 using a for-loop: https://youtu.be/pui8KXT_uPl.

Hints and Guidelines

We can solve the problem using a **for loop**, via which we will pass through the numbers from 1 to 100 using the **i** variable, and print the numbers in the body of the loop:



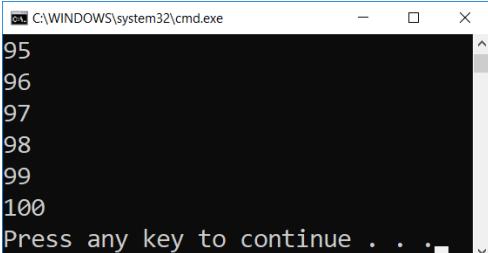
```

Loops - Microsoft Visual Studio
File Edit View Project Build Debug Team Tools Architecture Test Analyze
Window Help
Quick Launch (Ctrl+Q)
Program.cs * x
Numbers1To100 Numbers1To100.Program Main(string[] args)
namespace Numbers1To100
{
    class Program
    {
        static void Main(string[] args)
        {
            for (int i = 1; i <= 100; i++)
            {
                Console.WriteLine(i);
            }
        }
    }
}

```

The screenshot shows the Microsoft Visual Studio interface with the 'Program.cs' file open. The code defines a 'Program' class with a 'Main' method that prints integers from 1 to 100 using a for loop. The Solution Explorer on the right shows a single project named 'Numbers1To100' containing the 'Program.cs' file.

Start the program using [Ctrl+F5] and test it:



```

C:\WINDOWS\system32\cmd.exe
95
96
97
98
99
100
Press any key to continue . . .

```

The screenshot shows a command-line window titled 'cmd.exe' with the path 'C:\WINDOWS\system32'. It displays the output of the program, which consists of the numbers 95, 96, 97, 98, 99, and 100, each on a new line. Below the numbers, there is a prompt 'Press any key to continue . . .' indicating that the program has finished executing.

Testing in the Judge System

Test your solution here: <https://judge.softuni.org/Contests/Practice/Index/510#0>.

You should get **100 points** (fully accurate solution).

Example: Numbers up to 1000, Ending by 7

Write a program that finds all numbers within the range [1 ... 1000] that end in 7.

Video: Numbers 1...1000 Ending by 7

Watch the video lesson to learn how to print all numbers in the range [1...1000], ending by 7: <https://youtu.be/oFJ72d5GUoo>.

Hints and Guidelines

We can solve the problem by combining a **for loop** for passing through the numbers from 1 to 1000 and a **condition** to check if each of the numbers ends in 7. Of course, there are other solutions too, but let's solve the problem using a **loop + condition**:

```
for (int i = 1; i <= 1000; i++)
{
    if (i % 10 == 7)
    {
        // TODO: print i
    }
}
```

Testing in the Judge System

Test your solution here: <https://judge.softuni.org/Contests/Practice/Index/510#1>.

Example: All Latin Letters

Write a program that prints the letters of the English Alphabet: **a, b, c, ..., z**.

Video: Latin Letters

Watch the following video lesson to learn how to print the Latin letters using a for-loop: <https://youtu.be/EKNPt69wsFM>.

Hints and Guidelines

It is good to know that the **for loops** don't only work with numbers. We can solve the task by running a **for loop** that passes sequentially through all letters in the English alphabet:

```
Console.WriteLine("Latin alphabet:");
for (int letter = 'a'; letter <= 'z'; letter++)
{
    Console.WriteLine(" " + letter);
}
Console.WriteLine();
```

Testing in the Judge System

Test your solution here: <https://judge.softuni.org/Contests/Practice/Index/510#2>.

Code Snippet for the for Loop in Visual Studio

In software development, we regularly need to write loops, dozens of times a day. This is why in most development environments (IDE) there are **code snippets** for writing loops. An example of such snippet is the **snippet for for loop in Visual Studio**. Write **for** in the C# code editor in Visual Studio and **hit [Tab] twice**. Visual Studio will open a snippet for you and write a full **for loop**:



Try it yourselves, in order to master the skill of using the code snippet for **for loop** in Visual Studio.

Exercises: Loops (Repetitions)

Now that we are familiar with loops, it is time **to practice our newly acquired skills**, and as you know, this is achieved by a lot of code writing. Let's solve some **practical problems**.

Video: Chapter Summary

Watch this video to review what we learned in this chapter: https://youtu.be/4G_oSUcx9ko.

What We Learned in This Chapter?

We can repeat a code block using a **for loop**:

```
for (int i = 1; i <= 10; i++)
{
    Console.WriteLine("i = " + i);
}
```

We can read a sequence of **n** numbers from the console this way:

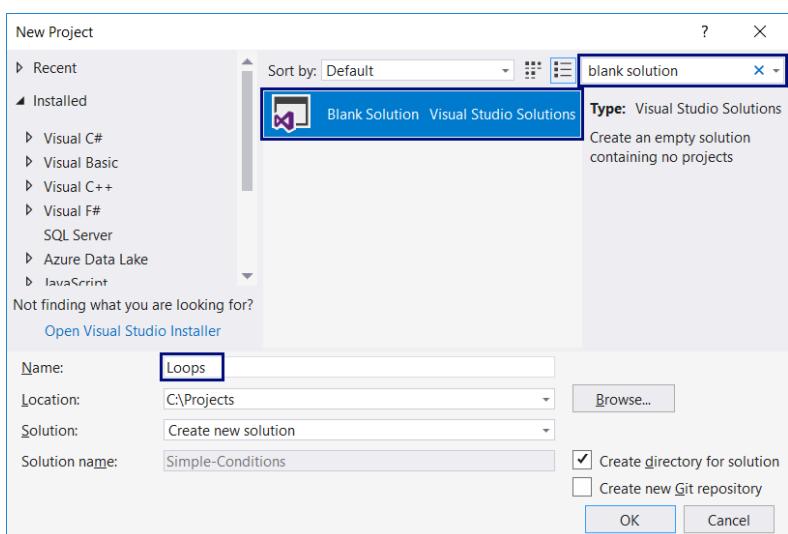
```
var n = int.Parse(Console.ReadLine());
for (int i = 0; i < n; i++)
{
    var num = int.Parse(Console.ReadLine());
}
```

Blank Solution in Visual Studio

At the start with the exercises, we will create a **Visual Studio solution** with the idea to hold the code for each exercises problem in a **separate C# project** inside the solution.

Create a **(Blank Solution)** in Visual Studio, as it is shown at the screenshot.

Set it up to **start the current project by default** (not the first one in the solution). Do that by right clicking on **Solution 'Loops'**, then on **[Set StartUp Projects...]**, the choose the option **[Current selection]**.



Problem: Summing up Numbers

Write a program that inputs n integers and sums them up.

- The first line of the input holds the number of integers n .
- Each of the following n lines holds an integer for summing.
- Sum up the numbers and finally print the result.

Sample Input and Output

Input	Output	Input	Output	Input	Output	Input	Output	Input	Output
2		3		4		1			
10		-10		45		999		0	
20	30	-20	-60	-20	43		999		0
		-30		7					
				11					

Video: Summing Numbers

Watch this video to learn how to sum numbers in for-loop: <https://youtu.be/t7PAichwl7k>.

Hints and Guidelines

We can solve the problem with summing up numbers in the following way:

- We read the input number n .
- We initially start with a sum $\text{sum} = 0$.
- We run a loop from 1 to n . On each step of the loop, we read the number num and add it to the sum sum .
- Finally, we print the calculated amount sum .

Below is the source code for the solution:

```
Console.WriteLine("n = ");
var n = int.Parse(Console.ReadLine());
Console.WriteLine("Enter the numbers:");
var sum = 0;

for (int i = 0; i < n; i++)
{
    var num = int.Parse(Console.ReadLine());
    sum = sum + num;
}

Console.WriteLine("sum = " + sum);
```

Testing in the Judge System

Test your solution here: <https://judge.softuni.org/Contests/Practice/Index/510#3>.

Problem: Max Number

Write a program that enters n integers ($n > 0$) and finds the **max number** among them (the largest). The first input line specifies the number of integers n . The next n lines hold the integers, one per line.

Sample Input and Output

Input	Output	Input	Output	Input	Output	Input	Output	Input	Output
2 100 99	100	3 -10 20 -30	20	4 45 -20 7 99	99	1 999	999	2 -1 -2	-1

Video: Largest Number

Watch this video lesson to learn how to find the largest number among a sequence of numbers:
<https://youtu.be/KErfOdOuezE>.

Hints and Guidelines

We will first enter one number **n** (the number of integers that are about to be entered). We assign the current maximum **max** an initial neutral value, for example **-1000000000000000** (or **int.MinValue**). Using a **for loop** that is iterated **n-1 times**, we read one integer **num** on each iteration. If the read number **num** is higher than the current maximum **max**, we assign the value of the **num** to the **max** variable. Finally, in **max** we must have stored the biggest number. We print the number on the console.

```
Console.WriteLine("n = ");
var n = int.Parse(Console.ReadLine());
var max = -1000000000000000;

for (int i = 1; i <= n; i++)
{
    var num = int.Parse(Console.ReadLine());
    if (num > max)
    {
        max = num;
    }
}

Console.WriteLine("max = " + max);
```

Testing in the Judge System

Test your solution here: <https://judge.softuni.org/Contests/Practice/Index/510#4>.

Problem: Min Number

Write a program that inputs **n** integers (**n > 0**) and finds **the min** number among them (the smallest number). First enter the number of integers **n**, then **n** numbers additionally, one per line.

Video: Smallest Number

Watch this video lesson to learn how to find the smallest number among a sequence of numbers:
<https://youtu.be/lHuz-mXbhzg>.

Sample Input and Output

Input	Output
1 50	50

Input	Output
2 100 99	99

Input	Output
3 -10 20 -30	-30

Input	Output
4 45 -20 7 99	-20

Hints and Guidelines

The problem is completely identical to the previous one, except this time we will start with another neutral starting value.

```
Console.WriteLine("n = ");
var n = int.Parse(Console.ReadLine());
var min = 1000000000000000;

for (int i = 1; i <= n; i++)
{
    var num = int.Parse(Console.ReadLine());
    if (num < min)
    {
        min = num;
    }
}

Console.WriteLine("min = " + min);
```

Testing in the Judge System

Test your solution here: <https://judge.softuni.org/Contests/Practice/Index/510#5>.

Problem: Left and Right Sum

Write a program that converts $2 * n$ integers and checks whether **the sum of the first n integers** (left sum) equals the **sum of the second n numbers** (right sum). In case the sums are equal, print "Yes" + **the sum**, otherwise print "No" + **the difference**. The difference is calculated as a positive number (by absolute value). The format of the output must be identical to the one in the examples below.

Sample Input and Output

Input	Output
2 10 90 60 40	Yes, sum = 100

Input	Output
2 90 9 50 50	No, diff = 1

Video: Left and Right Sum

Watch this video lesson to learn how to calculate the left and the right sum and their difference: https://youtu.be/s_uAugTnC8w.

Hints and Guidelines

We will first input the number **n**, after that the first **n** numbers (**left half**) and sum them up. We will then proceed with inputting more **n** numbers (**the right half**) and sum them up. We calculate the **difference** between the sums by absolute value: `Math.Abs(leftSum - rightSum)`. If the difference is 0, print "Yes" + the sum, otherwise print "No" + the difference.

```
Console.WriteLine("n = ");
var n = int.Parse(Console.ReadLine());
var leftSum = 0;
var rightSum = 0;

for (int i = 0; i < n; i++)
{
    leftSum = leftSum + int.Parse(Console.ReadLine());
}
// TODO: read and calculate the rightSum

if (leftSum == rightSum)
{
    Console.WriteLine("Yes, sum = " + leftSum);
}
else
{
    var difference = Math.Abs(rightSum - leftSum);
    Console.WriteLine("No, diff = " + difference);
}
```

Testing in the Judge System

Test your solution here: <https://judge.softuni.org/Contests/Practice/Index/510#6>.

Problem: Even / Odd Sum

Write a program that inputs **n** integers and checks whether **the sum of the numbers on even positions** is equal to **the sum of the numbers on odd positions**. In case the sums are equal, print "Yes" + the sum, otherwise, print "No" + the difference. The difference is calculated by absolute value. The format of the output must be identical to the examples below.

Sample Input and Output

Input	Output	Input	Output	Input	Output
4		4		3	
10		3		5	No
50	Yes	5		1	Diff = 1
60	Sum = 70	1		-2	
20					

Video: Even / Odd Sum

Watch this video to learn how to sum the element at even and odd positions and how to calculate their equality or difference: <https://youtu.be/79QsS7FI2qg>.

Hints and Guidelines

We input the numbers one by one and calculate the two **sums** (of the numbers on **even** positions and the numbers on **odd** positions). Identically to the previous problem, we calculate the absolute value of the difference and print the result ("Yes" + **the sum** in case of difference of 0 or "No" + **the difference** in any other case).

```
Console.WriteLine("n = ");
var n = int.Parse(Console.ReadLine());
var oddSum = 0;
var evenSum = 0;

for (int i = 0; i < n; i++)
{
    var element = int.Parse(Console.ReadLine());
    if (i % 2 == 0)
    {
        evenSum += element;
    }
    else
    {
        oddSum += element;
    }
}
// TODO: print the sum / difference
```

Testing in the Judge System

Test your solution here: <https://judge.softuni.org/Contests/Practice/Index/510#7>.

Problem: Sum of Vowels

Write a program that inputs **text** (string), calculates and prints **the sum of the values of vowels** according to the table below:

a	e	i	o	u
1	2	3	4	5

Sample Input and Output

Input	Output	Comments
hello	6	e+o = 2+4 = 6
hi	3	(i = 3)

Input	Output	Output
bamboo	9	a+o+o = 1+4+4 = 9
beer	4	e+e = 2+2 = 4

Video: Sum of Vowels

Watch this video lesson to learn how to sum the vowels in a text: <https://youtu.be/9hi9G3vRAOU>.

Hints and Guidelines

We read the input text `s`, null the sum and run a loop from `0` to `s.Length - 1` (text length - 1). We check each letter `s[i]` and verify if it is a vowel, and accordingly, add its value to the sum.

```
var s = Console.ReadLine();
var sum = 0;

for (int i = 0; i < s.Length; i++)
{
    if (s[i] == 'a')
    {
        sum += 1;
    }
    else if (s[i] == 'e')
    {
        sum += 2;
    }
    else if (s[i] == 'i')
    {
        sum += 3;
    }
    else if (s[i] == 'o')
    {
        sum += 4;
    }
    else if (s[i] == 'u')
    {
        sum += 5;
    }
}

Console.WriteLine("Vowels sum = " + sum);
```

Testing in the Judge System

Test your solution here: <https://judge.softuni.org/Contests/Practice/Index/510#8>.

Problem: Element Equal to the Sum of the Rest

Write a program that inputs `n` integers and checks whether among them there is a number equal to the sum of all the rest. If there is such an element, print "Yes" + its value, otherwise – "No" + the difference between the largest element and the sum of the rest (by absolute value).

Sample Input and Output

Input	Output	Comments	Input	Output	Comments
7 3 4 1 1 2 12 1	Yes Sum = 12	$3 + 4 + 1 + 2 + 1 + 1 = 12$	3 1 1 10	No Diff = 8	$ 10 - (1 + 1) = 8$

Input	Output	Input	Output	Input	Output
3 1 1 1	No Diff = 1	3 5 5 1	No Diff = 1	4 6 1 2 3	Yes Sum = 6

Hints and Guidelines

We must calculate the **sum** of all elements, find the **largest** of them and check the condition.

Testing in the Judge System

Test your solution here: <https://judge.softuni.org/Contests/Practice/Index/510#9>.

Problem: Even / Odd Positions

Write a program that reads **n numbers** and calculates the **sum**, the **min** and **max** values of the numbers on **even** and **odd** positions (counted from 1). If there is no min / max element, print "No".

Sample Input and Output

Input	Output
6 2 3 5 4 2 1	OddSum=9, OddMin=2, OddMax=5, EvenSum=8, EvenMin=1, EvenMax=4

Input	Output
2 1.5 -2.5	OddSum=1.5, OddMin=1.5, OddMax=1.5, EvenSum=-2.5, EvenMin=-2.5, EvenMax=-2.5

Input	Output
1 1	OddSum=1, OddMin=1, OddMax=1, EvenSum=0, EvenMin>No, EvenMax>No

Input	Output
3 -1 -2 -3	OddSum=-4, OddMin=-3, OddMax=-1, EvenSum=-2, EvenMin=-2, EvenMax=-2

Input	Output
1 -5	OddSum=-5, OddMin=-5, OddMax=-5, EvenSum=0, EvenMin>No, EvenMax>No

Input	Output
5 3 -2 8 11 -3	OddSum=8, OddMin=-3, OddMax=8, EvenSum=9, EvenMin=-2, EvenMax=11

Hints and Guidelines

The task combines some of the previous tasks: finding the **min** and **max** value and **sum**, as well as processing of elements on **even and odd positions**. Check them out.

In the current task it is better to work with **fractions** (not integers). The sum, the min and the max value will also be fractions. We must use a **neutral starting value** upon finding the min/max value, for example **1000000000.0** or **-1000000000.0**. If the end result is the neutral value, we will print "No".

Testing in the Judge System

Test your solution here: <https://judge.softuni.org/Contests/Practice/Index/510#10>.

Problem: Equal Pairs

There are **2 * n numbers**. The first and the second number form a **pair**, the third and the fourth number also, and so on. Each pair has a **value** – the sum of its numbers. Write a program that checks if all pairs have equal value.

In case the value is the same, print "Yes, value=..." + the value, otherwise, print the **maximum difference** between two neighboring pairs in format: "No, maxdiff=..." + the maximum difference.

The input consists of the number **n**, followed by **2*n integers**, all of them one per line.

Sample Input and Output

Input	Output
3	Yes, value=3
1	
2	Comments
0	
3	
4	
-1	values = {3, 3, 3} equal values

Input	Output
2	No, maxdiff=1
1	Comments
2	
2	values = {3, 4} differences = {1}
2	max difference = 1

Input	Output
2	No, maxdiff=2
-1	Comments
2	
0	values = {2, 4, 4, 0} differences = {2, 0, 4}
-1	max difference = 4

Input	Output
	Yes, value=10
	Comments
1	
5	
5	values = {10} one value equal values

Input	Output
2	
-1	
0	Yes, value=-1
0	
-1	

Input	Output
4	
1	
1	
3	
1	No, maxdiff=4
2	
2	
0	
0	

Hints and Guidelines

We read the input numbers **in pairs**. For each pair we calculate its **sum**. While reading the input pairs, for each pair except the first one, we must calculate **the difference compared to the previous one**. In order to do that, we need to store as a separate variable the sum of the previous pair. Finally, we find the **largest difference** between two pairs. If it is 0, print "Yes" + the value, otherwise – "No" + the difference.

Testing in the Judge System

Test your solution here: <https://judge.softuni.org/Contests/Practice/Index/510#11>.

Lab: Turtle Graphics GUI Application

In the current chapter we learned about **loops** as a programming construction that allows to repeat a particular action or a group of actions multiple times. Now let's play with them. In order to do that, we will draw some figures that will consist of a large number of repeating graphical elements, but this time we will not do it on the console, but in a graphical environment using "**turtle graphics**". It will be interesting. And it is not hard at all. Try it!

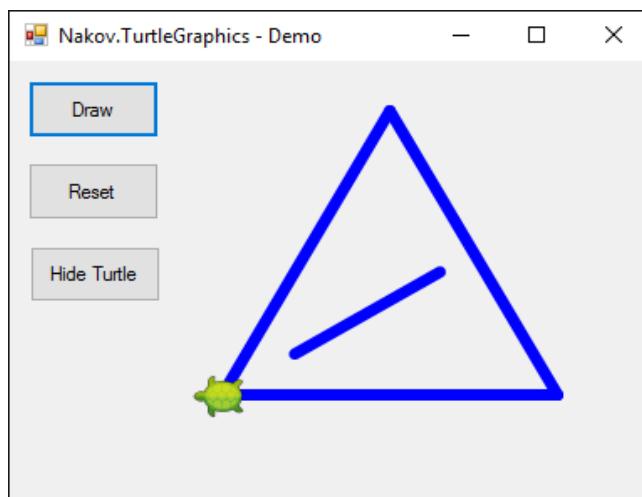
Video: Turtle Graphics

Watch the video to learn about turtle graphics and how to draw figures by moving and rotating the turtle in a Windows Forms GUI application: <https://youtu.be/WwSjMHo0Fx4>.

What Shall We Build?

The purpose of the following exercise is to play with a "move and rotate" **drawing library**, also known as "**turtle graphics**". We will build a graphical application (GUI App) in which we will **draw various**

figures by moving our "turtle" across the screen via operations like "move 100 positions ahead", "turn 30 degrees to the right", "move 50 more positions ahead". The app will look approximately like this:



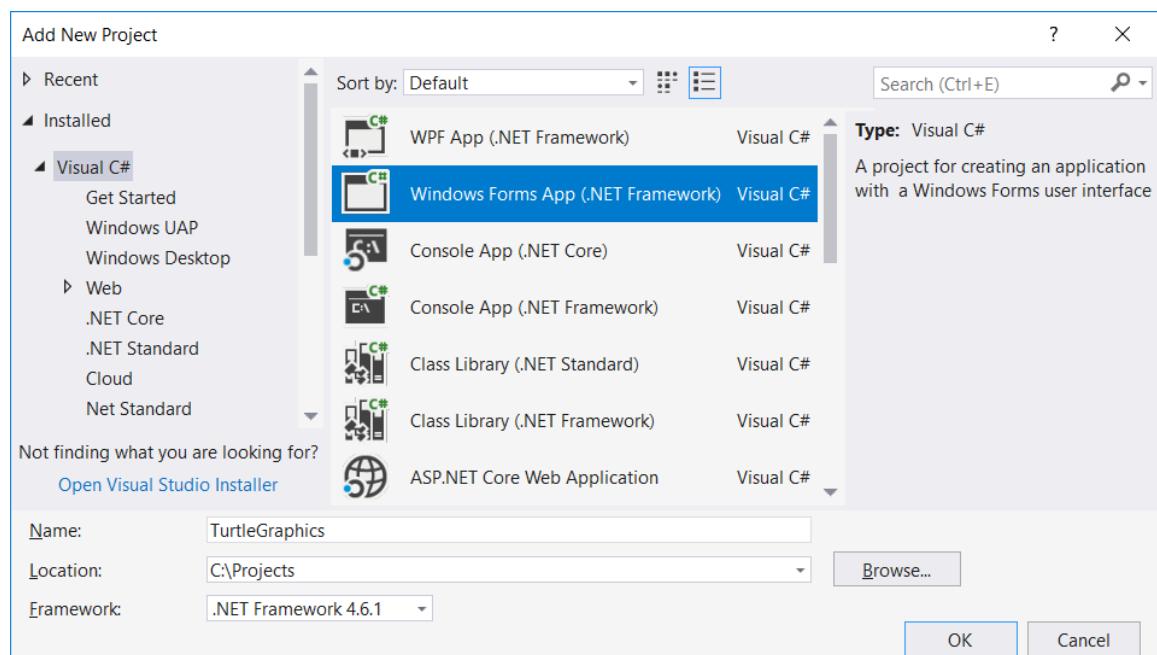
Turtle Graphics – Concepts

Let's first get familiar with the concept of drawing "Turtle Graphics". Take a look at the following sources:

- Definition of "turtle graphics": <http://c2.com/cgi/wiki?TurtleGraphics>
- Article on "turtle graphics" in Wikipedia: https://en.wikipedia.org/wiki/Turtle_graphics
- Interactive online tool for drawing with a turtle: <https://blockly-games.appspot.com/turtle>

Creating a New C# Project

We will start by creating a new C# Windows Forms Project:

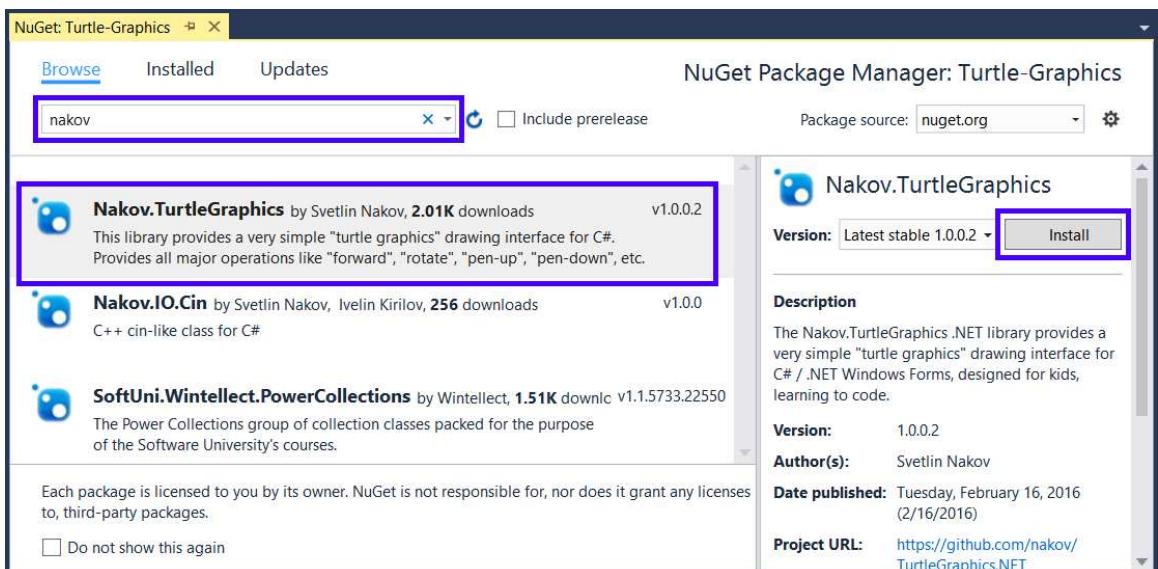


Installing Turtle Graphics NuGet Package

Install the NuGet package "**Nakov.TurtleGraphics**" to your new Windows Forms project. From Visual Studio you can add **external libraries**(packages) to an existing C# project. They add up additional functionality to our applications. The official repository for C# libraries is maintained by Microsoft and is called **NuGet** (<http://www.nuget.org>).

Right-click the **Solution Explorer** project and select [**Manage NuGet Packages...**] like it is shown on the screenshot.

A **NuGet** package search and installation window will open. Let's search for packages by keyword **nakov**. A few packages will be found. Select **Nakov.TurtleGraphics**. Click [**Install**] to install it to your C# project:

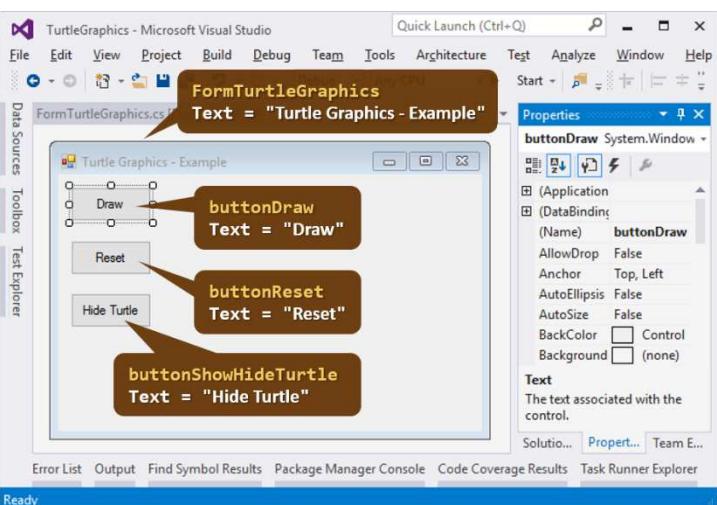


The external library **Nakov.TurtleGraphics** is already included in our C# project. It defines **Turtle** class that represents a drawing **turtle**. In order to use it, add (**Form1.cs**) in the C# code for our form. Add the following code at the top of the file:

```
using Nakov.TurtleGraphics;
```

Adding the Buttons

Now we need to add **three buttons** into the form and change their **names** and **properties** as it is shown on the screenshot.



Implementing the [Draw] Button

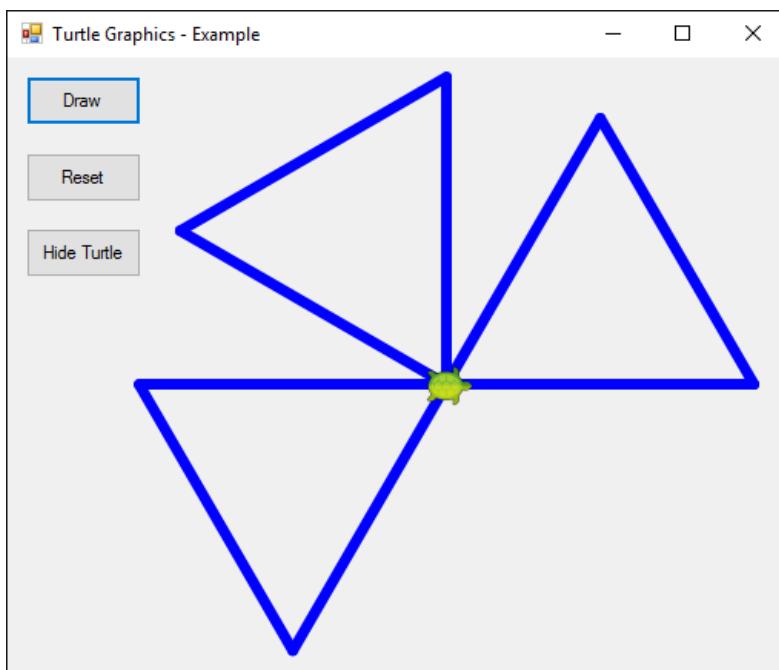
Double-click the [Draw] button in order to add the code that will be executed upon its pressing. Write the following code:

```
private void buttonDraw_Click(object sender, EventArgs e)
{
    Turtle.Rotate(30);
    Turtle.Forward(200);
    Turtle.Rotate(120);
    Turtle.Forward(200);
    Turtle.Rotate(120);
    Turtle.Forward(200);
}
```

This code moves and rotates the turtle that is initially in the center of the screen (in the middle of the form) and draws an equilateral triangle. You can edit it and play with it.

Testing the Application

Start the application by pressing [Ctrl+F5]. Test if it works (press the [Draw] button a few times):



Adding Complexity to the Turtle Drawing Code

Now you can modify the `turtle` code and make it more complex:

```
// Assign a delay to visualize the drawing process
Turtle.Delay = 200;

// Draw a equilateral triangle
Turtle.Rotate(30);
```

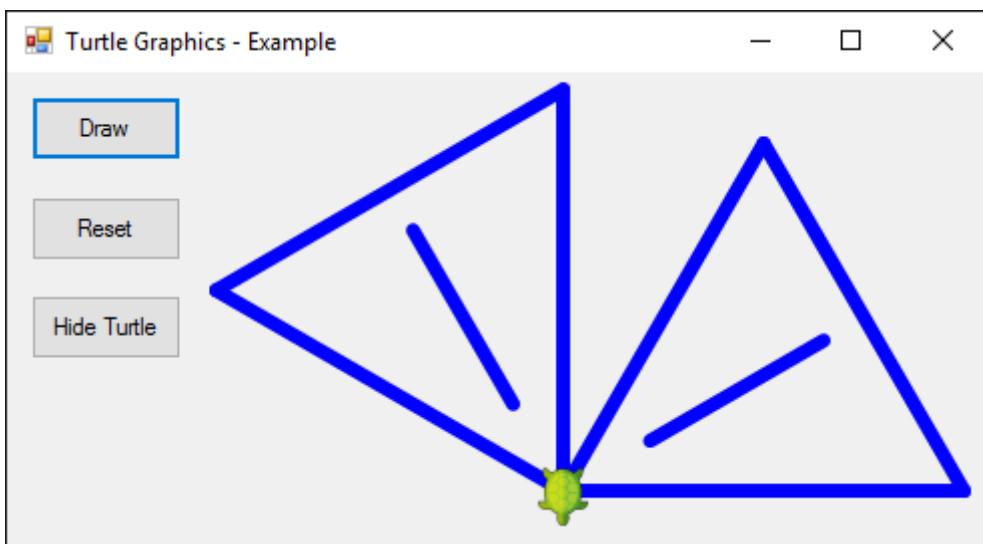
```

Turtle.Forward(200);
Turtle.Rotate(120);
Turtle.Forward(200);
Turtle.Rotate(120);
Turtle.Forward(200);

// Draw a line in the triangle
Turtle.Rotate(-30);
Turtle.PenUp();
Turtle.Backward(50);
Turtle.PenDown();
Turtle.Backward(100);
Turtle.PenUp();
Turtle.Forward(150);
Turtle.PenDown();
Turtle.Rotate(30);

```

Start the application again by pressing [Ctrl+F5]. Test whether the new turtle program works:



Now our turtle draws more complex figures via a nice animated motion.

Implementing the [Reset] Button

Now let's write the code for the other two buttons. The purpose of the [Reset] button is to delete the graphics and to start drawing from the beginning:

```

private void buttonReset_Click(object sender, EventArgs e)
{
    Turtle.Reset();
}

```

Implementing the [Show / Hide Turtle] Buttons

The purpose of the [Show / Hide Turtle] button is to show or hide the turtle:

```

private void buttonShowHideTurtle_Click(object sender, EventArgs e)
{
    if (Turtle.ShowTurtle)
    {
        Turtle.ShowTurtle = false;
        this.buttonShowHideTurtle.Text = "Show Turtle";
    }
    else
    {
        Turtle.ShowTurtle = true;
        this.buttonShowHideTurtle.Text = "Hide Turtle";
    }
}

```

Once again, **start** the application by [Ctrl+F5] and test whether it works correctly.

Exercises: Turtle Graphics

Now, it is your time to draw a few figures with the turtle, using sequences of moves and rotations. Add a new **additional button** for drawing each of the figures, shown below.

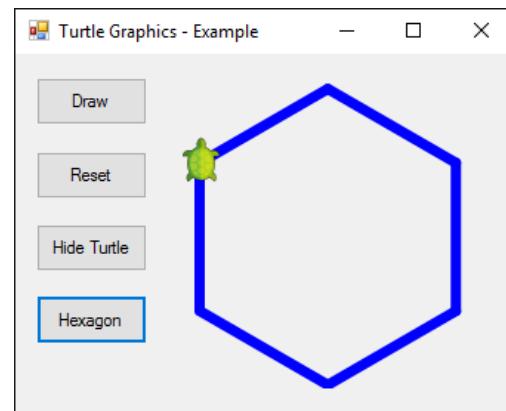
Problem: * Draw a Hexagon with the Turtle

Add a **[Hexagon]** button that draws a **regular hexagon**. It may look like at the screenshot.

Hint:

Repeat 6 times the following in a loop:

- 60 degrees rotation.
- Forward step of 100.



Problem: * Draw a Star with the Turtle

Add a **[Star]** button that draws a **star with 5 beams (pentagram)** in green color, as on the screenshot.

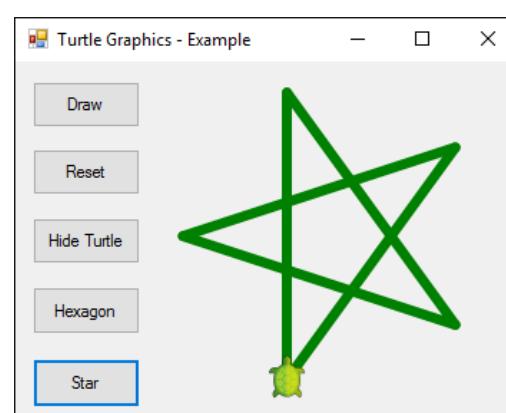
Hints:

Change the color to green:

```
Turtle.PenColor = Color.Green.
```

Repeat 5 times the following in a loop:

- Forward step of 200.
- 144 degrees rotation.



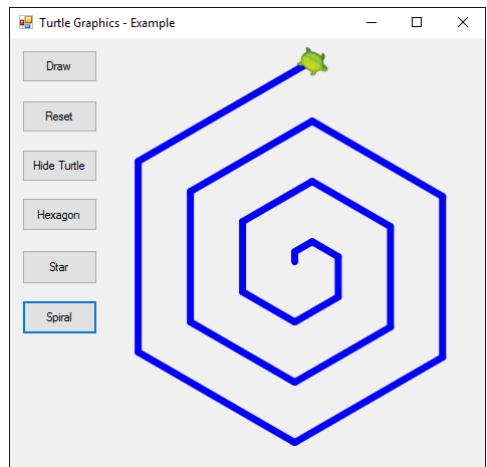
Problem: * Draw a Spiral with the Turtle

Add a [Spiral] button that draws a spiral with 20 beams, as on the figure.

Hints:

- Draw in a loop by moving ahead and rotating.
- In each step, decrease gradually the length of the forward step and rotate at 60 degrees.

Can you solve the same problem differently?



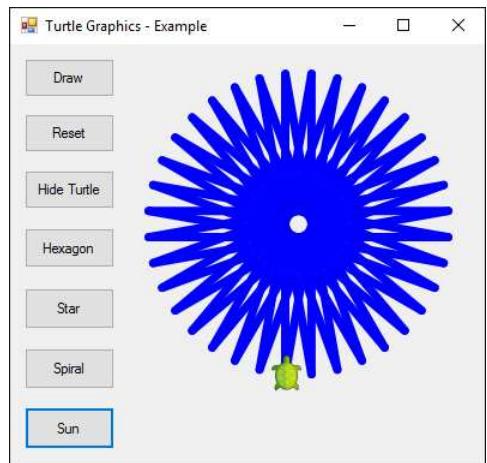
Problem: * Draw a Sun with the Turtle

Add a [Sun] button that draws a sun with 36 beams, as on the figure.

Hints:

- The entire circle consists of 360 degrees = $36 * 10$ degrees
- Think about how many times you will move the turtle forward, then rotate, then move the turtle back, etc.

Can you solve the same problem differently?



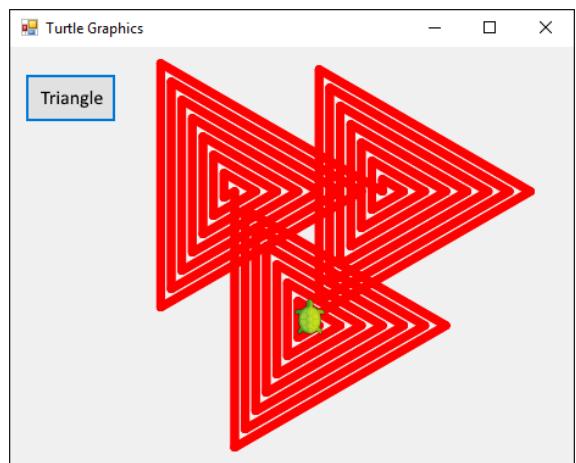
Problem: * Draw a Spiral Triangle with the Turtle

Add a [Triangle] button that draws three triangles with 22 beams each, as on the figure.

Hint:

Draw in a loop by moving forward and rotating. In each step, increase the length of the forward step with 10 and rotate 120 degrees. Repeat 3 times for the three triangles.

If you have a problem with the above exercises, you can ask for help in the SoftUni official discussion forum (<http://forum.softuni.org>) or in the SoftUni official Facebook page (<https://fb.com/softuni.org>).



High-quality education profession and job for software engineers

- ✓ Free start for **absolute beginners**
- ✓ Choose among the **most in-demand** professions in the software industry
- ✓ Assistance for **career start**
- ✓ **Top trainers**, mentors and active learning **community**



SoftUni provides practical, modern and **innovative education** for programming, IT, design, marketing and digital skills for thousands of young people. The "**Software University**" program builds true **software engineering professionals**.

Join the end-to-end **software engineering learning program** at SoftUni to master the most in-demand programming concepts, languages, software technologies and skills through a modern teaching methodology, **interactive learning platform** and tons of practical exercises and projects. Our **curriculum** is developed together with the **IT companies** to closely match the industry demands.

SoftUni works directly with the **companies** from the software industry to **find a job** for its students and to make them successful software engineers.

The path of the student at SoftUni



Apply now

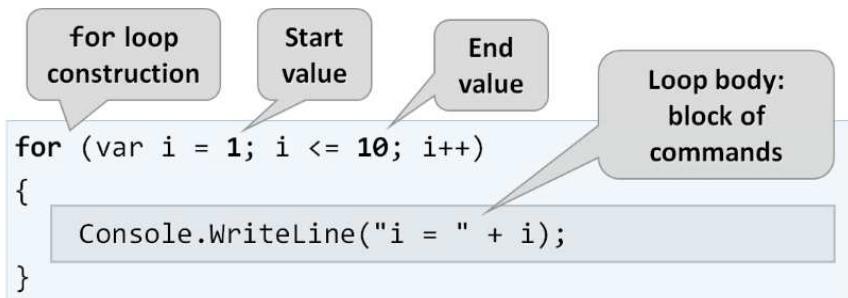
softuni.org/apply

Chapter 5.2. Loops – Exam Problems

In the previous chapter we learned how to run a block of commands **more than once**. That's why we introduced a **for loop** and reviewed its main use cases. The purpose of this chapter is to improve our knowledge by solving some more complex problems with loops, used for exams. For some of them we will show examples of detailed solutions, and for others we will leave only guidance.

For Loops – Quick Review

Before we start working, it will be good to review again the **for loop** construction:



For loops consist of:

- Initialization block in which the variable-counter is declared (`var i`) and its initial value is set.
- Repeat condition (`i <= 10`), executing once, before each iteration of the loop.
- Restarting the counter (`i++`) – this code is executed after each iteration.
- Body of the loop – contains random block of source code.

Now, after the review, let's solve a few **problems** with loops from [exams in SoftUni](#).

Problem: Histogram

We have **n integer numbers** within the range of [1 ... 1000]. Some percent of them **p1** are under 200, another percent **p2** are from 200 to 399, percent **p3** are from 400 to 599, percent **p4** are from 600 to 799 and the rest **p5** percent are from 800 upwards. Write a program that calculates and prints the percentages **p1, p2, p3, p4** and **p5**.

Example: we have **n = 20** numbers: 53, 7, 56, 180, 450, 920, 12, 7, 150, 250, 680, 2, 600, 200, 800, 799, 199, 46, 128, 65. We get the following distribution and visualization:

Group	Numbers	Numbers Count	Percent
< 200	53, 7, 56, 180, 12, 7, 150, 2, 199, 46, 128, 65	12	$p1 = 12 / 20 * 100 = 60.00\%$
200... 399	250, 200	2	$p2 = 2 / 20 * 100 = 10.00\%$
400... 599	450	1	$p3 = 1 / 20 * 100 = 5.00\%$
600... 799	680, 600, 799	3	$p4 = 3 / 20 * 100 = 15.00\%$
≥ 800	920, 800	2	$p5 = 2 / 20 * 100 = 10.00\%$

Input Data

On the first line of the input there is an integer n ($1 \leq n \leq 1000$) that represents the count of lines of numbers that will be passed. On each of the following n lines we have **one integer** within range of [1 ... 1000] – numbers, on which we have to calculate the histogram.

Output Data

Print on the console a **histogram that consists of 5 lines**, each of them containing a number within the range of [0% ... 100%], formatted up to two digits after the decimal point (for example 25.00%, 66.67%, 57.14%).

Sample Input and Output

Input	Output	Input	Output
9	33.33%	14	57.14%
367	33.33%	53	14.29%
99	11.11%	7	7.14%
200	11.11%	56	14.29%
799	11.11%	180	7.14%
999		450	
333		920	
555		12	
111		7	
9		150	
		250	
		680	
		2	
		600	
		200	

Input	Output	Input	Output	Input	Output
3	66.67%	4	75.00%	7	14.29%
1	0.00%	53	0.00%	800	28.57%
2	0.00%	7	0.00%	801	14.29%
999	0.00%	56	0.00%	250	14.29%
	33.33%	999	25.00%	199	28.57%

Hints and Guidelines

We can split the program that solves this problem into three parts:

- **Reading the input data** – in the current problem this includes reading of the number n , followed by n count of integers, each on a single line.
- **Processing the input data** – in this case that means allocating the numbers into groups and calculating the percentage breakdown by groups.
- **Printing the output** – printing the histogram on the console in the specified format.

Before we proceed, we will make a small deviation from the current topic, namely, we will briefly mention that in programming every variable has a certain **data type**. In this problem we will use the numeral types **int** for **integers** and **double** for **real numbers**. Often, to make it easier, programmers miss the explicit specification of the type by replacing it with the keyword **var**. For better understanding we will write the type upon declaring the variables.

We will now proceed with the implementation of each of the above points.

Reading the Input Data

Before we proceed to reading the input data, we must **declare the variables**, in which we will store it. This means choosing the right data type and appropriate names.

```
int n;                                // Variables to keep the count of numbers by groups
// Variables to keep the distribution in percentages int cntP1 = 0;
// for each group                         int cntP2 = 0;
double p1Percentage = 0;                int cntP3 = 0;
double p2Percentage = 0;                int cntP4 = 0;
double p3Percentage = 0;                int cntP5 = 0;
double p4Percentage = 0;
double p5Percentage = 0;
```

In the variable **n** we will store the count of numbers that we will read from the console. We choose **int type**, because **n** is an **integer** within the range from 1 to 1000. For the variables in which we will store the percentages, we choose **double type**, because **they are not expected** to always be **integers**. Additionally, we declare the variables **cntP1**, **cntP2** etc., in which we will keep the count of the numbers from the respective group, and we choose again **int type**.

After we have declared the needed variables, we can read the number **n** from the console:

```
n = int.Parse(Console.ReadLine());
```

Distributing Numbers in Groups

To read and distribute each number in its corresponding group, we will use a **for loop** from 0 to **n** (count of numbers). Every iteration of the loop will read and distribute **one single** number (**currentNumber**) into its corresponding group. In order to determine if a number belongs to a group, we **perform a check in its respective range**. If the above is true, we increase the count of the numbers in the corresponding group (**cntP1**, **cntP2** etc.) by 1.

```
for (int i = 0; i < n; i++)
{
    int currentNumber = int.Parse(Console.ReadLine());

    if (currentNumber < 200)
    {
        cntP1++;
    }
    else if (currentNumber >= 200 && currentNumber <= 399)
    {
        cntP2++;
    }
    else if (currentNumber >= 400 && currentNumber <= 599)
    {
```

```

        cntP3++;
    }
    else if (currentNumber >= 600 && currentNumber <= 799)
    {
        cntP4++;
    }
    else
    {
        cntP5++;
    }
}

```

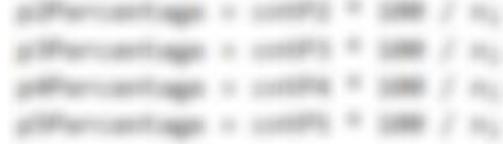
Calculating Percentages

After we have determined how many numbers there are in each group, we can move on to calculating the percentages, which is the main purpose of the problem. For this we will use the following formula:

$$(\text{group percentage}) = (\text{count of numbers in group}) * 100 / (\text{count of all numbers})$$

This formula in the program code looks like this:

```
p1Percentage = cntP1 * 100 / n;
```



If we divide by **100** (**int** number type) instead of **100.0** (**double** number type), we will perform the so-called **integer division** and the variable will save only the whole part of the division, and this is not the result we want. For example: $5 / 2 = 2$, but $5 / 2.0 = 2.5$. Considering this, the formula for the first variable will look like this:

```
p1Percentage = cntP1 * 100.0 / n;
// TODO: calculate the other percentages
```

To make it even clearer, let's take a look at the example on the right.

In this case **n = 3**. For the loop we have:

- **i = 0** – we read the number 1, which is less than 200 and falls into the first group (**p1**) and increase the group count (**cntP1**) by 1.
- **i = 1** – we read the number 2, which again falls into the first group (**p1**) and increase its count (**cntP1**) again by 1.
- **i = 2** – we read the number 999, which falls into the last group (**p5**), because its bigger than 800, and increase the count of the group (**cntP5**) with 1.

Input	Output
3	66.67%
1	0.00%
2	0.00%
999	33.33%

After reading the numbers in group **p1** we have 2 numbers, and in **p5** we have 1 number. We have **no numbers** in the other groups. By applying the above formula, we calculate the percentages of each group. If we multiply in the formula by **100**, instead of **100.0** we will obtain for group **p1** 66%, and for group **p5** – 33% (without fractional part).

Printing the Output

We only have to print the results output. The description says that the percentages must be with **precision of two points after the decimal point**. To achieve this, write “**:f2**” after the placeholder:

```
Console.WriteLine("{0:f2}%", p1Percentage);
```



Testing in the Judge System

Test your solution here: <https://judge.softuni.org/Contests/Practice/Index/511#0>.

Problem: Smart Lilly

Lilly is **N** years old. For each **birthday** she receives a present. For each **odd** birthday (1, 3, 5, ..., n) she receives **toys**, and for each **even** birthday (2, 4, 6, ..., n) she receives **money**. For her **second birthday** she received **10.00 USD**, and the amount is increased by **10.00 USD** for each following even birthday (2 -> 10, 4 -> 20, 6 -> 30 etc.). Over the years Lilly has secretly saved her money. Lilly's **brother**, in the years when she **received money**, took **1.00 USD** from each of the amounts. Lilly **has sold the toys**, received over the years, **each one for P USD** and added the sum to the amount of saved money. With the money she wanted to **buy a washing machine** for **X USD**.

Write a program that calculates **how much money she has saved** and if it is enough **to buy a washing machine**.

Input Data

We read from the console **3 numbers**, each on a separate line:

- Lilly's **age** – **integer** in the range of [1 ... 77].
- **Price of the washing machine** – **number** in the range of [1.00 ... 10 000.00].
- **Unit price of each toy** – **integer** in the range of [0 ... 40].

Output Data

Print on the console one single line:

- If Lilly's money is enough:
 - “Yes! {N}” – where **N** is the remaining money after the purchase
- If the money is not enough:
 - “No! {M}” – where **M** is the insufficiency amount
 - Numbers **N** and **M** must be **formatted up to the second digit after the decimal point**.

Sample Input and Output

Input	Output	Comments
21 1570.98 3	No! 997.98	She has saved 550 USD . She has sold 11 toys 3 USD each = 33 USD . Her brother has taken for 10 years 1 USD each year = 10 USD . Remaining amount: $550 + 33 - 10 = 573$ USD. $573 < 1570.98$: she did not manage to buy a washing machine. The insufficiency amount is $1570.98 - 573 = 997.98$ USD.

Input	Output	Comments
10 170.00 6	Yes! 5.00	For the first birthday she gets a toy; 2nd -> 10 USD; 3rd -> toy; 4th -> $10 + 10 = 20$ USD; 5th -> toy; 6th -> $20 + 10 = 30$ USD; 7th -> toy; 8th -> $30 + 10 = 40$ USD; 9th -> toy; 10th -> $40 + 10 = 50$ USD. She has saved -> $10 + 20 + 30 + 40 + 50 = 150$ USD. She sold 5 toys for 6 USD each = 30 USD. Her brother took 1 USD 5 times = 5 USD. Remaining amount -> $150 + 30 - 5 = 175$ USD. $175 \geq 170$ (price of the washing machine): she managed to buy it and is left with $175 - 170 = 5$ USD.

Hints and Guidelines

The solution of this problem, like the previous one, can also be split into three parts – **reading** the input data, **processing** them and **printing the output**.

Reading the Input Data

```
int age = int.Parse(Console.ReadLine());
double priceOfWashingMachine = [REDACTED]
int presentPrice = [REDACTED]
```

We start again by selecting the appropriate **data types** and names of variables. For the Lilly's years (**age**) and the unit price of the toy (**presentPrice**) the description requires **integers**. That's why we will use **int type**. For the price of the washing machine (**priceOfWashingMachine**) we know that it is **real number** and we choose **double**. Of course, we can skip the explicit specification of type, by using **var**. In the above code we **declare** and **initialize** (assign value to) the variables.

Creating Helper Variables

To solve the problem, we will need several helper variables – for the **count of toys** (**numberOfToys**), for the **saved money** (**savedMoney**) and for the **money received on each birthday** (**moneyForBirthday**). The initial value of **moneyForBirthday** is 10, because the description says that the first sum, which Lilly gets, is 10 USD:

```
int numberOfToys = 0;
int savedMoney = 0;
int moneyForBirthday = 10;
```

Calculating Savings

```
for (int currentYear = 1; currentYear <= age; currentYear++)
{
    if (currentYear % 2 == 0)
    {
        savedMoney += (moneyForBirthday - 1);
        moneyForBirthday += 10;
    }
    else
    {
        numberOfToys++;
    }
}
```

```
}
```

With a **for** loop we iterate through every Lilly's birthday. When the leading variable is an **even number**, that means that Lilly **has received money** and we add this money to her total savings. At the same time, we **subtract 1 USD** – the money that her brother took. Then we **increase** the value of the variable **moneyForBirthday**, i.e. we increase by 10 the sum that she will receive on her next birthday. On the contrary, when the leading variable is an **odd number**, we increase the count of **toys**. We do the parity check by **division with remainder (%) by 2** – when the remainder is 0, the figure is even, and in case of remainder 1 – it is odd.

We also add the money from the sold toys to Lilly's savings.

```
savedMoney += numberOfToys * presentPrice;
```

Formatting and Printing the Output

Finally, we need to print the obtained results, considering the formatting specified in the description, i.e. sum needs to be rounded up to 2 symbols after the decimal point:

```
Console.WriteLine(savedMoney >= priceOfWashingMachine ?  
    $"Yes! {{savedMoney - priceOfWashingMachine}:{0.00}}"  
    : $"No! {{priceOfWashingMachine - savedMoney}:{0.00}}");
```

In this case we choose to use the **conditional operator** (`? :`) (also called ternary operator), because the record is shorter. Its syntax is as follows: `operand1 ? operand2 : operand3`. The first operand needs to be of **bool type** (i.e. to return **true/false**). If `operand1` returns **true**, `operand2` will be executed, and if it returns **false** – `operand3` will be executed. In our case we check if the **money saved** by Lilly is enough for a washing machine. If it is more than or equal to the price of a washing machine, the check `savedMoney >= priceOfWashingMachine` will return **true** and will print "Yes! ...", and if it is less – the result will be **false** and "No! ..." will be printed. Of course, instead of conditional operand, we can use **if** checks.

Learn more about the conditional operator from: <https://www.dotnetperls.com/ternary>, <https://msdn.microsoft.com/en-us/library/ty67wk28.aspx>.

Testing in the Judge System

Test your solution here: <https://judge.softuni.org/Contests/Practice/Index/511#1>.

Problem: Back to the Past

Ivan is **18 years old** and receives an inheritance that consists of **X** money and a **time machine**. He decides to return to **1800** but does not know **if the money will be enough** to live without working. Write a program that calculates if Ivan will have enough money to not have to work until a particular year (inclusive). Assuming that **for every even** (1800, 1802, etc.) year he **will spend 12 000 dollars**. For **every odd one** (1801, 1803, etc.) he will spend **12 000 + 50 * [the age he will have reached in the given year]**.

Input Data

The input is read from the console and **contains exactly 2 lines**:

- **Inherited money** – a real number in the range [1.00 ... 1 000 000.00].
- **Year, until which he has to live in the past (inclusive)** – integer number in the range [1801 ... 1900].

Output Data

Print on the console 1 line. The sum must be formatted up to the two symbols after the decimal point:

- If money is enough:
 - "Yes! He will live a carefree life and will have {N} dollars left." – where N is the money that will remain.
- If money is NOT enough:
 - "He will need {M} dollars to survive." – where M is the sum that is NOT enough.

Sample Input and Output

Input	Output	Explanations
50000 1802	Yes! He will live a carefree life and will have 13050.00 dollars left.	1800 → even → Spends 12000 dollars → Remain 50000 – 12000 = 38000 1801 → odd → Spends 12000 + 19*50 = 12950 dollars → Remaining 38000 – 12950 = 25050 1802 → even → Spends 12000 dollars → Remaining 25050 – 12000 = 13050
100000.15 1808	He will need 12399.85 dollars to survive.	1800 → even → Remaining 100000.15 – 12000 = 88000.15 1801 → odd → Remaining 88000.15 – 12950 = 75050.15 ... 1808 → odd → -399.85 - 12000 = -12399.85 12399.85 is not enough

Hints and Guidelines

Let's solve the problem step by step: read the input data, iterate over the years, check the heritage and print the output.

Reading the Input Data

The method to solve this task is no different than the previous ones, so we start **declaring and initializing** the necessary variables:

```
double heritage = Console.ReadLine();
int yearToLive = Console.ReadLine();
int years = 18;
```

The requirements say that Ivan is 18 years old, so when declaring the variable **years** we assign it an initial value of **18**. We read the other variables from the console.

Iterating through the Years

Using a **for loop**, we will iterate through all years. We **start from 1800** – the year in that Ivan returns, and we reach the **year until which he must live in the past**. We check in the loop if the current year is

even or odd. We do this by division with remainder (%) by 2. If the year is even, we subtract from heritage 12000, and if is odd, we subtract from heritage $12000 + 50 * (\text{years})$.

```
for (int currentYear = 1800; currentYear <= yearToLive; currentYear++)
{
    if (currentYear % 2 == 0)
    {
        heritage -= 12000;
    }
    else
    {
        heritage -= (12000 + 50 * years);
    }
    years++;
}
```

Checking for Enough Heritage and Printing the Output

Finally, we need to print out the results by checking if the heritage is enough to live without working or not. If the heritage is a positive number, we print: "Yes! He will live a carefree life and will have {N} dollars left.", and if it is a negative number: "He will need {M} dollars to survive.". Do not forget to format the sum up to the second digit after the decimal point.

Hint: Consider using the `Math.Abs(...)` function when printing the output, if the heritage is not enough.

Testing in the Judge System

Test your solution here: <https://judge.softuni.org/Contests/Practice/Index/511#2>.

Problem: Hospital

For a certain period of time, patients arrive at the hospital every day for an examination. It has initially 7 doctors. Each doctor can treat only one patient per day, but sometimes there is a shortage of doctors, so the remaining patients are sent to other hospitals. Every third day the hospital makes calculations and if the count of untreated patients is greater than the count of treated ones, another doctor is appointed. Appointment takes place before the daily patient acceptance begins.

Write a program, that calculates for a given period of time, the count of treated and untreated patients.

Input Data

Input is read from the console and contains:

- On the first line – the period, for which you need to make calculations. Integer in the range of [1 ... 1000].
- On the next lines (equal to the count of days) – count of the patients, who arrive for treatment for the current day. Integer in the range of [0 ... 10 000].

Output Data

Print on the console 2 lines:

- On the first line: “Treated patients: {count of treated patients}.”
- On the second line: “Untreated patients: {count of untreated patients}.”

Sample Input and Output

Input	Output	Comments
4 7 27 9 1	Treated patients: 23. Untreated patients: 21.	Day 1: 7 treated and 0 untreated patients for the day Day 2: 7 treated and 20 untreated patients for the day Day 3: By this moment the treated patients are 14, and untreated ones – 20 –> New doctor is appointed. –> 8 treated and 1 untreated patients for the day Day 4: 1 treated and 0 untreated patients for the day Total: 23 treated and 21 untreated patients.

Input	Output	Input	Output
3 7 7 7	Treated patients: 21. Untreated patients: 0.	6 25 25 25 25 25 2	Treated patients: 40. Untreated patients: 87.

Hints and Guidelines

Let's solve the problem step by step: read the input data, calculate the number treated and untreated patients and print the output.

Reading the Input Data

Again, we begin by **declaring and initializing** the required variables:

```
int period = int period = ReadInt();
```

```
int treatedPatients = 0;
int untreatedPatients = 0;
int countOfDoctors = 7;
```

The period in which we have to make the calculations is read from the console and saved in the **period** variable. We will also need some helper variables: the number of treated patients (**treatedPatients**), the number of untreated patients (**untreatedPatients**) and the number of doctors (**countOfDoctors**), which is initially 7.

Calculating the Treated and Untreated Patients

```
for (int day = 1; day <= period; day++)
{
    var currentPatients = int currentPatients = ReadInt();
```

```

if ((day % 3 == 0) && (untreatedPatients > treatedPatients))
{
    countOfDoctors++;
}

if (currentPatients > countOfDoctors)
{
    treatedPatients += countOfDoctors;
    untreatedPatients += currentPatients - countOfDoctors;
}
else
{
    treatedPatients += currentPatients;
}
}

```

With the help of a **for** loop we iterate through all days in the given period (**period**). For each day, we read from the console the number of the patients (**currentPatients**). Increasing doctors by requirements can be done **every third day**, BUT only if the count of untreated patients is **greater** than the count of treated ones. For this purpose, we check if the day is third one – with the arithmetical operator for division with remainder (%): **day % 3 == 0**.

For example:

- If the day is **third one**, the remainder of the division by **3** will be **0** (**3 % 3 = 0**) and the check **day % 3 == 0** will return **true**.
- If the day is **second one**, the remainder of the division by **3** will be **2** (**2 % 3 = 2**) and the check will return **false**.
- If the day is **forth one**, the remainder of the division will be **1** (**4 % 3 = 1**) and the check will return again **false**.

If **day % 3 == 0** returns **true**, the system will check whether the count of untreated patients is greater than the count of treated ones: **untreatedPatients > treatedPatients**. If the result is again **true**, then the count of doctors will be increased (**countOfDoctors**).

Then we check if the count of the patients for the day (**currentPatients**) is greater than the count of doctors (**countOfDoctors**). If the count of the patients is **greater**:

- Increase the value of the variable **treatedPatients** by the count of doctors (**countOfDoctors**).
- Increase the value of the variable **untreatedPatients** by the count of the remaining patients, which we calculate by subtracting the count of doctors from the count of patients (**currentPatients - countOfDoctors**).

If the count of patients is **not greater**, increase only the variable **treatedPatients** with the count of patients for the day (**currentPatients**).

Finally, we need to print the count of treated and count of untreated patients.

Testing in the Judge System

Test your solution here: <https://judge.softuni.org/Contests/Practice/Index/511#3>.

Problem: Division without Remainder

We have n integers in the range of [1 ... 1000]. Among them, some percentage p_1 are divisible without remainder by 2, percentage p_2 are divisible without remainder by 3, percentage p_3 are divisible without remainder by 4. Write a program that calculates and prints the p_1 , p_2 and p_3 percentages.

Example: We have $n = 10$ numbers: 680, 2, 600, 200, 800, 799, 199, 46, 128, 65. We get the following distribution and visualization:

Division without remainder by	Numbers	Count	Percent
2	680, 2, 600, 200, 800, 46, 128	7	$p_1 = (7 / 10) * 100 = 70.00\%$
3	600	1	$p_2 = (1 / 10) * 100 = 10.00\%$
4	680, 600, 200, 800, 128	5	$p_3 = (5 / 10) * 100 = 50.00\%$

Input Data

On the first line of the input is the integer n ($1 \leq n \leq 1000$) – count of numbers. On each of the next n lines we have **one integer** in the range of [1 ... 1000] – numbers that needs to be checked for division.

Output Data

Print on the console **3 lines**, each of them containing a percentage between 0% and 100%, two digits after the decimal point, for example 25.00%, 66.67%, 57.14%.

- On the **first line** – percentage of the numbers that are **divisible by 2**.
- On the **second line** – percentage of the numbers that are **divisible by 3**.
- On the **third line** – percentage of the numbers that are **divisible by 4**.

Sample Input and Output

Input	Output	Input	Output	Input	Output
10 680 2 600 200 800 799 199 46 128 65	70.00% 10.00% 50.00%	3 3 6 9	33.33% 100.00% 0.00%	1 12	100.00% 100.00% 100.00%

Hints and Guidelines

For the current and for the next problem you will need to write by yourself the program code, following the given guidelines.

The program that solves the current problem is similar to the one from the **Histogram** problem, which we reviewed earlier. That's why we can start declaring the required variables. Sample names of variables may be: **n** – count of numbers (that we need to read from the console) and **divisibleBy2**, **divisibleBy3**, **divisibleBy4** – helper variables that keeps the count of the numbers in the corresponding group.

To read and allocate each number to its corresponding group we have to rotate **for loop** from **0** to **n** (count of numbers). Each iteration of the loop should read and allocate **one single number**. The difference here is that a **single number can get into several groups at once**, so we have to make **three separate if checks for each number** – respectively check whether it is divided by 2, 3 and 4 (using **if-else** statement in this case will not work, because after finding a match it interrupts further checking of conditions) and increase the value of the variable that keeps the count of numbers in the corresponding group.

Finally, you need to print the obtained results, by following the specified format.

Testing in the Judge System

Test your solution here: <https://judge.softuni.org/Contests/Practice/Index/511#4>.

Problem: Logistics

You are responsible for the logistics various types of cargo. Depending on the weight of each cargo, you need a **different vehicle**, and this will cost a **different price per ton**:

- Up to **3 tons** – a **minibus** (200 USD per ton).
- From **over 3 and up to 11 tons** – **truck** (175 USD per ton).
- **Over 11 tons** – **train** (120 USD per ton).

Your task is to calculate **the average price per tone of the cargo**, and also **what percentage of the cargo is transported in each vehicle**.

Input Data

From the console we must read a **sequence of numbers**, each on a separate line:

- First line: **count of cargo** for transportation – **integer** in the range of [1 ... 1000].
- On the next lines we pass **the tonnage of the current cargo** – **integer** in the range of [1 ... 1000].

Output Data

Print on the console **4 lines**, as follows:

- Line #1 – **the average price per tone of the cargo** (rounded up to the second digit after the decimal point).
- Line #2 – **percentage** of the cargo, carried by **minibus** (between 0.00% and 100.00%, rounded up to the second digit after the decimal point).
- Line #3 – **percentage** of the cargo, carried by **truck** (between 0.00% and 100.00%).
- Line #4 – **percentage** of the cargo, carried by **train** (between 0.00% and 100.00%).

Sample Input and Output

The example below demonstrates and explains the computation process:

Input	Output	Explanations
4	143.80	By minibus you transport two of the cargo 1 + 3 , total of 4 tons.
1	16.00%	By truck you transport one of the cargo: 5 tons.
5	20.00%	By train you transport one of the cargo: 16 tons.
16	64.00%	Sum of all cargo is: $1 + 5 + 16 + 3 = 25$ tons. Percentage of the cargo by minibus : $4/25 * 100 = 16.00\%$ Percentage of the cargo by truck : $5/25 * 100 = 20.00\%$ Percentage of the cargo by train : $16/25 * 100 = 64.00\%$ Average price per ton of carried cargo: $(4 * 200 + 5 * 175 + 16 * 120) / 25 = 143.80$
3		

Hints and Guidelines

First, we will read the weight of each cargo and sum how much tons are being transported by minibus, truck and train, and we will calculate the total tons of the transported cargo. We will calculate prices for each type of transportation according to the transported tons and the total price. Finally, we will calculate and print the total average price per ton

Input	Output
5	149.38
2	7.50%
10	42.50%
20	50.00%
1	
7	

Input	Output
4	120.35
53	0.00%
7	0.63%
56	99.37%
999	

and how much of the cargo is being transported by different types of transport in percentages.

We declare the needed variables, for example: **countOfLoads** – count of the cargos for transportation (we read them from the console), **sumOfTons** – sum of the tonnage of all cargos, **microbusTons**, **truckTons**, **trainTons** – variables that keeps the sum of the cargo tonnage, transported by minibus, truck and train.

We still need a **for loop** from **0** to **countOfLoads-1**, to iterate through all cargo types. For each cargo we read its weight (in tons) from the console and save it in a variable, for example **tons**. We add to the tonnage the sum of all cargo (**sumOfTons**) the weight of the current cargo (**tons**). Once we have read the weight of the current cargo, we need to determine which vehicle type will be used (minibus, truck or train). For this we will need **if-else** statements:

- If the value of the variable **tons** is less than **3**, increase the value of **microbusTons** by the value of **tons**: **microbusTons += tons;**
- Otherwise, if the **tons** are less than **11**, increase **truckTons** by **tons**.
- If **tons** are more than **11**, increase **trainTons** by **tons**.

Before we print the output, we need to calculate the percentage of tons, transported by each vehicle and the average price per ton. For the average price per ton we will declare one more helper variable **totalPrice**, in which we will sum the total price of all transported cargo (by minibus, truck and train). We will calculate an average price, by dividing **totalPrice** of **sumOfTons**. You need to calculate by yourself the percentages of tons, transported by each vehicle, and print the results, keeping the format specified in the description.

Testing in the Judge System

Test your solution here: <https://judge.softuni.org/Contests/Practice/Index/511#5>.

Chapter 6.1. Nested Loops

In the current chapter we will be looking at **nested loops** and how to use **for** loops to **draw** various **figures on the console**, that contain symbols and signs, ordered in rows and columns on the console. We will use **single** and **nested loops** (loops that stay in other loops), **calculations** and **checks**, in order to print on the console simple and not so simple figures by specified sizes.

Video: Chapter Overview

Watch a video lesson to learn what we will learn in this chapter: <https://youtu.be/sbmhyr1Yz7U>.

Introduction to Nested Loops by Examples

In programming **loops can be nested**, which means that in a loop we can put another loop. This is an example of **nested for-loops**, which are used to draw a square of **n** rows, each holding **n** times the chars **=**:

```
int n = int.Parse(Console.ReadLine());
for (int row = 1; row <= n; row++)
{
    for (int col = 1; col <= n; col++)
    {
        Console.Write("=");
    }
    Console.WriteLine();
}
```

Run the above code example: <https://repl.it/@nakov/nested-for-loops-draw-square-csharp>.

If we run the above code and enter **5** as input, the **output** will be as follows:

```
5
=====
=====
=====
=====
=====
```

Using a combination of **calculations**, **conditional statements** and **nested loops**, we can implement **more complex logic**. For example, we can draw a **rhombus of stars** as follows:

```
int n = 10;
for (int row = 1; row < n; row++)
{
    var spaces = Math.Abs(n / 2 - row);
    var stars = n/2 - spaces;
    for (int col = 1; col <= spaces; col++)
        Console.Write(" ");
    for (int col = 1; col <= stars; col++)
        Console.Write("* ");
    Console.WriteLine();
}
```

Run the above code example: <https://repl.it/@nakov/nested-for-loops-draw-rhombus-csharp>.

The above code will print on the console the following **output**:

```

*
* *
* * *
* * * *
* * * * *
* * * *
* * *
* *
*
```

Let's explain in greater detail how to use **nested loops** to implement more complex logic in our C# programs.

Nested Loops – Concepts

A **nested loop** is a construction where **in the body of one loop** (outer one) **stays another loop** (inner one). In each iteration of the outer loop, **the whole inner loop** is executed. This happens in the following way:

- When nested loops start executing, **the outer loop starts** first: the controlling variable is **initialized** and after a check for ending the loop the code in its body is executed.
- After that, **the inner loop is executed**. The controlling variables start position is initialized, a check for ending the loop is made and the code in its body is executed.
- When reaching the specified value for **ending the loop**, the program goes back one step up and continues executing the previous (outer) loop. The controlling variable of the outer loop changes with one step, a check is made to see if the condition for ending the loop is met and **a new execution of the nested (inner) loop is started**.
- This is repeated until the variable of the outer loop meets the condition to **end the loop**.

Video: Nested Loops

Watch this video to learn the concepts of nested loops: <https://youtu.be/Sj49u6XlzXU>.

Nested Loops – Examples

Here is an **example** that illustrates nested loops. The aim is again to print a rectangle made of **n * n** stars, in which for each row a loop iterates from **1** to **n**, and for each column a nested loop is executed from **1** to **n**:

```

var n = int.Parse(Console.ReadLine());
for (var r = 1; r <= n; r++)
{
    for (var c = 1; c <= n; c++)
    {
        Console.Write("*");
    }
    Console.WriteLine();
}
```

```

5
*****
*****
*****
*****
*****
Press any key to continue . . .
```

If we enter **5** as input on the console, the above sample code will print the output shown on the right.

Let's look at the example above. After initializing **the first (outer) loop**, its **body**, which contains **the second (nested) loop** starts executing. By itself it prints on one row **n** number of stars. After **the inner loop finishes** executing at the first iteration of the outer one, **the first loop will continue**, i.e. it will print an empty row on the console. **After that**, the variable of **the first loop** will be **renewed** and the whole **second** loop will be executed again. The inner loop will execute as many times as the body of the outer loop executes, in this case **n** times.

Example: Rectangle Made of 10 x 10 Stars

Print on the console a rectangle made out of **10 x 10** stars.

Video: Rectangle of 10 x 10 Stars

Watch this video lesson to learn how to print a rectangle of **10 x 10** stars on the console: <https://youtu.be/XNsgT4yqws>.

Hints and Guidelines

```
for (var i = 1; i <= 10; i++)
{
    Console.WriteLine(new string('*', 10));
}
```

How does the example work? We initialize **a loop with a variable `i = 1`**, which increases with each iteration of the loop, while it is **less or equal to 10**. This way the code in the body of the loop is **executed 10 times**. In the body of the loop we print a new line on the console `new string('*', 10)`, which creates a string of 10 stars.

The above solution uses a trick to avoid nesting loops: it prints 10 stars using the string constructor (instead of printing a star 10 times in a nested loop). Another solution, using **nested for-loops**, might look like this:

```
for (int row = 1; row <= 10; row++)
{
    for (int col = 1; col <= 10; col++)
        Console.Write('*');
    Console.WriteLine();
}
```

Testing in the Judge System

Test your solution here: <https://judge.softuni.org/Contests/Practice/Index/512#0>.

Example: Rectangle Made of N x N Stars

Write a program that enters a positive integer **n** and prints a rectangle made out of **N x N** stars:

Input	Output	Input	Output	Input	Output
2	** **	3	*** *** ***	4	**** **** **** ****

Video: Rectangle of N x N Stars

Watch this video lesson to learn how to print a rectangle of $N \times N$ stars on the console using nested for-loops: <https://youtu.be/9sB4Z2Tl1AE>.

Hints and Guidelines

This is sample solution, which uses a single loop, holding a command to print n stars:

```
int n = int.Parse(Console.ReadLine());
for (int i = 1; i <= n; i++)
{
    Console.WriteLine(
        new string('*', n));
}
```

You may also use **nested for-loops**: outer loop $1..n$ for the **rows** and inner loop $1..n$ for the **columns**, which prints a single star.

Testing in the Judge System

Test your solution here: <https://judge.softuni.org/Contests/Practice/Index/512#1>.

Examples: Square Made of Stars

Print on the console a square made of $N \times N$ stars (use a space between the stars, on the same line):

Input	Output
2	* *

Input	Output
3	* * * * * * * * *

Input	Output
4	* * * * * * * * * * * * * * * *

Hints and Guidelines

The problem is similar to the last one. The difference here is that we need to figure out how to add a whitespace after the stars so that there aren't any excess white spaces in the beginning or the end.

```
var n = int.Parse(Console.ReadLine());
for (var r = 1; r <= n; r++)
{
    Console.Write("*");
    for (var c = 1; c < n; c++)
    {
        Console.WriteLine(" *");
    }
    Console.WriteLine();
}
```

Testing in the Judge System

Test your solution here: <https://judge.softuni.org/Contests/Practice/Index/512#2>.

Example: Triangle Made of Dollars

Write a program that takes an integer n and prints a triangle made of dollars of size n .

Input	Output
2	\$ \$ \$

Input	Output
3	\$ \$ \$ \$ \$ \$

Input	Output
4	\$ \$ \$ \$ \$ \$ \$ \$ \$ \$

Video: Triangle of Dollars

Watch this video to learn how to print a triangle of dollars on the console, using nested for-loops: <https://youtu.be/Pbfe1FOnMNE>.

Hints and Guidelines

The problem is **similar** to those for drawing a **rectangle** and **square**. Once again, we will use **nested loops**, but there is a **catch** here. The difference is that **the number of columns** that we need to print depends on **the row**, on which we are and not on the input number n . From the example input and output data we see that **the count of dollars depends** on which **row** we are on at the moment of the printing, i.e. 1 dollar means first row, 3 dollars mean third row and so on. Let's see the following example in detail. We see that **the variable of the nested loop** is connected with the variable of **the outer one**. This way our program prints the desired triangle.

```
for (var row = 1; row <= n; row++)
{
    Console.WriteLine("$");
    for (var col = 1; col < row; col++)
    {
        Console.Write(" ");
    }
    Console.WriteLine();
}
```

Testing in the Judge System

Test your solution here: <https://judge.softuni.org/Contests/Practice/Index/512#3>.

Example: Square Frame

Write a program that takes an integer n and draws on the console a **square frame** with a size of $n * n$.

Input	Output
4	+ - - + - - - - + - - +

Input	Output
5	+ - - - + - - - - - - - - - + - - - +

Input	Output
6	+ - - - - + - - - - - - - - - - - - - - - - + - - - - +

Video: Square Frame

Watch this video lesson to learn how to print a square frame on the console using nested loops:
<https://youtu.be/LS2uqvggfSA>.

Hints and Guidelines

We can solve the problem in the following way:

- We read from the console the number **n**.
- We print **the upper part**: first a **+** sign, then **n-2** times **-** and in the end a **+** sign.
- We print **the middle part**: we print **n-2** rows, as we first print a **|** sign, then **n-2** times **-** and in the end again a **|** sign. We can do this with nested loops.
- We print **the lower part**: first a **+** sign, then **n-2** times **-** and in the end a **+** sign.

Here is an example implementation of the above idea with nested loops:

```
int n = int.Parse(Console.ReadLine());  
  
// Print the top row: + - - - +  
Console.Write("+");  
for (int i = 0; i < n - 2; i++)  
{  
    Console.Write(" -");  
}  
Console.WriteLine(" +");  
  
// Print the mid rows: | - - - |  
for (int row = 0; row < n - 2; row++)  
{  
    Console.Write("|");  
    for (int i = 0; i < n - 2; i++)  
    {  
        Console.Write(" -");  
    }  
    Console.WriteLine(" |");  
}  
  
// Print the bottom row: + - - - +  
Console.Write("+");  
for (int i = 0; i < n - 2; i++)  
{  
    Console.Write(" -");  
}  
Console.WriteLine(" +");
```

Testing in the Judge System

Test your solution here: <https://judge.softuni.org/Contests/Practice/Index/512#4>.

Exercises: Drawing Figures

Let's look at how to **draw figures** using **nested loops** with more complex logic, for which we need to think more before coding.

Video: Chapter Summary

Watch this video to review what we learned in this chapter: <https://youtu.be/tj4s3BOijO4>.

What We Learned in This Chapter?

Before starting, let's review what we learned in this chapter.

We became acquainted with the **new string** constructor:

```
string printMe = new string('*', 5);
```

We learned to draw figures with nested **for** loops:

```
for (var r = 1; r <= 5; r++)
{
    Console.Write("*");
    for (var c = 1; c < 5; c++)
        Console.Write(" *");
    Console.WriteLine();
}
```

Problem: Rhombus Made of Stars

Write a program that takes a positive integer **n** and prints a **rhombus made of stars** with size **n**.

Input	Output	Input	Output	Input	Output	Input	Output
1	*	2	* * **	3	* * * * * * * * *	4	* * * * * * * * * * * * * * *

Video: Rhombus of Stars

Watch this video lesson to learn how to print a rhombus of stars on the console using nested loops: <https://youtu.be/BaSgBU6yLU8>.

Hints and Guidelines

To solve this problem, we need to mentally **divide the rhombus** into **two parts** – **upper** one, which **also** includes the middle row, and **lower** one. For the **printing** of each part we will use **two** separate loops, as we leave the reader to decide the dependency between **n** and the variables of the loops. For the first loop we can use the following guidelines:

- We print **n-row** white spaces.

- We print *.
- We print **row-1** times *.

The second (lower) part will be printed **similarly**, which again we leave to the reader to do.

```
for (var row = 1; row <= n; row++)
{
    for (var col = 1; col <= n - row; col++)
    {
        Console.Write(" ");
    }
    Console.WriteLine("*");
    for (var col = 1; col < row; col++)
    {
        Console.Write(" *");
    }
    Console.WriteLine();
}
// TODO: print the down side of the rhombus
```

Testing in the Judge System

Test your solution here: <https://judge.softuni.org/Contests/Practice/Index/512#5>.

Problem: Christmas Tree

Write a program that takes a number **n** ($1 \leq n \leq 100$) and prints a Christmas tree with height of **n+1**.

Input	Output	Input	Output	Input	Output	Input	Output
1	* *	2	* *	3	* * ** ** *** ***	4	* * ** ** *** *** **** ****

Video: Christmas Tree

Watch this video lesson to learn how to print a Christmas tree on the console using nested loops: <https://youtu.be/UecoBfhUlkk>.

Hints and Guidelines

From the examples we see that the Christmas tree can be divided into three logical parts. The first part is the stars and the white spaces before and after them, the middle part is |, and the last part is again stars, but this time there are white spaces only before them. The printing can be done with only one loop and the **new string(...)** constructor, which we will use once for the stars and once for the white spaces.

```
int n = int.Parse(Console.ReadLine());
for (int i = 0; i <= n; i++)
{
    var stars = new string('*', i);
```

```

var spaces = new string(' ', n - i);
Console.WriteLine(spaces);
Console.Write(stars);
Console.Write(" | ");
Console.Write(stars);
Console.WriteLine(spaces);
}

```

Testing in the Judge System

Test your solution here: <https://judge.softuni.org/Contests/Practice/Index/512#6>.

Problem: Sunglasses

Write a program that takes an integer n ($3 \leq n \leq 100$) and prints sunglasses with size of $5*n \times n$ as found in the examples:

Input	Output
3	***** * ***** *////* */// ***** * *****

Input	Output
4	***** * ***** *////* */// *////* */// ***** * *****

Input	Output
5	***** * ***** *////* */// *////* */// *////* */// ***** * *****

Video: Sunglasses

Watch this video lesson to learn how to print sunglasses on the console using nested loops: <https://youtu.be/MTQhldgno4k>.

Hints and Guidelines

From the examples we can see that the sunglasses can be divided into **three parts** – upper, middle and lower one. A part of the code with which the problem can be solved is given below.

Printing the Top and Bottom Rows

When drawing the upper and lower rows we need to print $2 * n$ stars, n white spaces and $2 * n$ stars.

```

// Print the top part
Console.WriteLine(new string('*', 2 * n));
Console.Write(new string(' ', n));
Console.WriteLine(new string('*', 2 * n));
for (int i = 0; i < n - 2; i++)
{

```

```
// TODO: print the middle part
}
// Print the bottom part
Console.WriteLine(new string('*', 2 * n));
Console.WriteLine(' ', n);
Console.WriteLine(new string('*', 2 * n));
```

Printing the Middle Rows

When drawing the middle part, we need to check if the row is $(n-1) / 2 - 1$, because in the examples we can see that in this row we need to print pipes instead of white spaces.

```
// Print the middle part
for (int i = 0; i < n - 2; i++)
{
    // TODO: print *////////*
    if (i == (n - 1) / 2 - 1)
        Console.WriteLine(' ', n);
    else
        Console.WriteLine(" " + " ".PadLeft(n));
    // TODO: print *////////*
    Console.WriteLine();
}
```

Testing in the Judge System

Test your solution here: <https://judge.softuni.org/Contests/Practice/Index/512#7>.

Problem: House

Write a program that takes a number n ($2 \leq n \leq 100$) and prints a house with size $n \times n$, just as in the examples:

Input	Output	Input	Output	Input	Output	Input	Output
2	** 	3	_*_ *** *	4	-**- **** ** **	5	--*-- -***- ***** *** ***

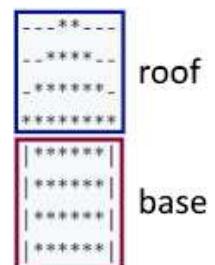
Video: Draw a House

Watch this video lesson to learn how to print a house on the console using nested loops: <https://youtu.be/ExjxRM0vXW4>.

Hints and Guidelines

We understand from the problem explanation that the house is with size of $n \times n$. What we see from the example input and output is that:

- The house is divided into two parts: roof and base.



- When **n** is an even number, the point of the house is "dull".
- When **n** is odd, **the roof** is one row larger than the **base**.

The Roof

- It comprises of **stars** and **dashes**.
- In the top part there are one or two stars, depending on if **n** is even or odd (also related to the dashes).
- In the lowest part there are many stars and no dashes.
- With each lower row, **the stars** increase by 2 and **the dashes** decrease by 2.

The Base

- The height is **n** rows.
- It is made out of **stars** and **pipes**.
- Each row comprises of 2 **pipes** – one in the beginning and one in the end of the row, and also **stars** between the pipes with string length of **n - 2**.

Reading the Input Data

We read **n** from the console and we save it in a variable of **int** type.

```
var n = int.Parse(Console.ReadLine());
```



It is very important to check if the input data is correct! In these tasks it is not a problem to directly convert the data from the console into **int** type, because it is said that we will be given valid integers. If you are making more complex programs it is a good practice to check the data. What will happen if instead of the character "A" the user inputs a number?

Calculating Roof Length

In order to draw **the roof**, we write down how many **stars** we start with in a variable called **stars**:

- If **n** is an even number, there will be 2 stars.
- If it is odd, there will be 1.

```
var stars = 1;
if (n % 2 == 0)
{
    stars++;
}
```

Calculate the length of **the roof**. It equals half of **n**. Write the result in the variable **roofLength**.

```
var roofLength = (int)Math.Ceiling(n / 2f);
```

It is important to note that when **n** is an odd number, the length of the roof is one row more than that of the **base**. In C# when you divide two numbers with a remainder, the result will be the number without remainder. Example:

```
int result = 3 / 2; // result 1
```

If we want to round up, we need to use the method `Math.Ceiling(...)`: `int result = (int)Math.Ceiling(3 / 2f);` In this example the division isn't between two integers. "f" after a number shows that this number is of `float` type (a floating-point number). The result of `3 / 2f` is `1.5f`. `Math.Ceiling(...)` rounds the division up. In this case `1.5f` will become `2`. `(int)` is used so that we can transfer the type back to `int`.

Printing the Roof

After we have calculated the length of the roof, we make a loop from 0 to `roofLength`. On each iteration we will:

- Calculate the number of `dashes` we need to draw. The number will be equal to `(n - stars) / 2`. We store it in a variable `padding`.

```
var padding = (n - stars) / 2;
```

- We print on the console: "dashes" (`padding / 2` times) + "stars" (`stars` times) + "dashes" (`padding / 2` times).

```
var line = new string('-', padding)
    + new string('*', stars)
    + new string('-', padding);
Console.WriteLine(line);
```

- Before the iteration is over, we add 2 to `stars` (the number of `the stars`).

```
stars += 2;
```



It is not a good idea to add many character strings as it is shown above, because this leads to `performance issues`. Learn more at: [https://en.wikipedia.org/wiki/String_\(computer_science\)#String_buffers](https://en.wikipedia.org/wiki/String_(computer_science)#String_buffers)

Printing the Base

After we have finished with the `roof`, it is time for `the base`. It is easier to print:

- We start with a loop from 0 to `n` (not inclusive).
- We print on the console: `| + * (n - 2 times) + |`.

```
for (int i = 0; i < n / 2; i++)
{
    var line = "|" + new string('*', n - 2) + "|";
    Console.WriteLine(line);
}
```

If you have written everything as it is here, the problem should be solved.

Testing in the Judge System

Test your solution here: <https://judge.softuni.org/Contests/Practice/Index/512#8>.

Problem: Diamond

Write a program that takes an integer `n` ($1 \leq n \leq 100$) and prints a diamond with size `n`, as in the examples below.

Input	Output	Input	Output	Input	Output	Input	Output	Input	Output
1	*	2	**	3	-*- *-* -*-	4	-*** *--* -***-	5	--*-- -*-* *---* -*-* --*--

Video: Draw a Diamond

Watch this video lesson to learn how to print a diamond on the console using nested loops:
<https://youtu.be/Z8crtxDztBk>.

Hints and Guidelines

What we know from the problem explanation is that the diamond is with size $n \times n$.

From the example input and output we can conclude that all rows contain exactly n symbols, and all the rows, with the exception of the top and bottom ones, have **2 stars**. We can mentally divide the diamond into 2 parts:

- **Upper** part. It starts from the upper tip down to the middle.
- **Lower** part. It starts from the row below the middle one and goes down to the lower tip (inclusive).

Upper Part

- If n is an **odd** number, it starts with **1 star**.
- If n is an **even** number, it starts with **2 stars**.
- With each row down, the stars get further away from each other.
- The space between, before and after **the stars** is filled up with **dashes**.

Lower Part

- With each row down, the stars get closer to each other. This means that the space (**the dashes**) between them is getting smaller and the space (**the dashes**) on the left and on the right is getting larger.
- The bottom-most part has 1 or 2 **stars**, depending on whether n is an even or odd number.

Upper and Lower Parts of the Diamond

- On each row, except the middle one, the stars are surrounded by inner and outer **dashes**.
- On each row there is space between the two **stars**, except on the first and the last row (sometimes **the star is 1**).

Reading the Input Data

We read n from the console and we save it in a variable of **int** type.

```
var n = int.Parse(Console.ReadLine());
```

Printing the Top Part of the Diamond

We start drawing the upper part of the diamond. The first thing we need to do is to calculate the number of the outer **dashes leftRight** (the dashes on the outer side of **the stars**). It is equal to $(n - 1) / 2$, rounded down.

```
var leftRight = (n - 1) / 2;
```

After we have calculated `leftRight`, we start drawing **the upper part** of the diamond. We can start by running a **loop** from `0` to `n / 2 + 1`(rounded down).

At each iteration of the loop the following steps must be taken:

- We draw on the console the left **dashes** (with length `leftRight`) and right after them the first **star**.

```
Console.WriteLine(new string('-', leftRight));
Console.Write("*");
```

- We will calculate the distance between the two **stars**. We can do this by subtracting from `n` the number of the outer **dashes**, and the number 2 (the number of **the stars**, i.e. the diamonds outline). We need to store the result of the subtraction in a variable `mid`.

```
var mid = n - 2 * leftRight - 2;
```

- If `mid` is lower than 0, we know that on the row there should be only 1 star. If it is higher or equal to 0 then we have to print **dashes** with length `mid` and one **star** after them.

- We draw on the console the right outer **dashes** with length `leftRight`.

```
Console.WriteLine(new string('-', leftRight));
```

- In the end of the loop we decrease `leftRight` by 1 (**the stars** are moving away from each other).

We are ready with the upper part.

Printing the Bottom Part of the Diamond

Printing the lower part is very similar to that of the upper part. The difference is that instead of decreasing `leftRight` with 1 in the end of the loop, we will increase `leftRight` with 1 at the beginning of the loop. Also, **the loop will be from 0 to $(n - 1) / 2$** .

```
var n = int.Parse(Console.ReadLine());
```



Repeating a code is considered **bad practice**, because the code becomes very hard to maintain. Let's imagine that we have a piece of code (e.g. the logic for drawing a row from the diamond) at a few more places and we decide to change it. For this we will have to go through all the places and change it everywhere. Now let's imagine that you need to reuse a piece of code not 1, 2 or 3 times but tens of times. A way to overcome this problem is to use **methods**. You can look for additional information for methods in the Internet or to look at [Chapter "10" \(Methods\)](#).

If we have written all correctly, then the problem is solved.

Testing in the Judge System

Test your solution here: <https://judge.softuni.org/Contests/Practice/Index/512#9>.

Lab: Drawing Ratings in Web

Now that we got used to **nested loops** and the way to use them to draw figures on the console, we can get into something even more interesting: we can see how loops can be used to **draw in a Web environment**. We will make a web application that visualizes a number rating (a number from 0 to

100) with stars. This kind of visualization is common in e-commerce sites, reviews of products, event rating, rating of apps, and others.

Don't worry if you don't understand all of the code, how exactly it is written and how the project works. It is normal, now we are learning to write code and we are a long way from the web development technologies. If you are struggling to write your project by following the steps, **watch the video** from the beginning of the chapter or ask for help in the SoftUni official **discussion forum** (<http://forum.softuni.org>) or in the SoftUni official **Facebook page** (<https://fb.com/softuni.org>).

Video: Building a Web App "Draw Ratings"

Watch a video lesson to learn how to build ASP.NET MVC Web app to draw ratings by given number: <https://youtu.be/ErnapuBjvZQ>.

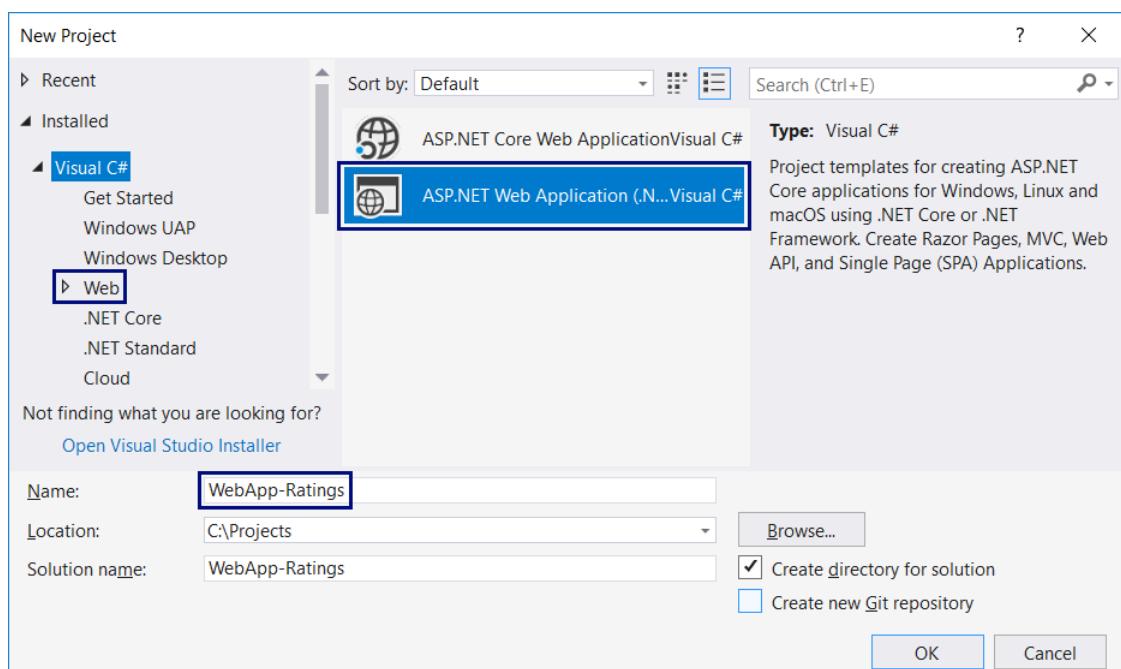
Ratings – Visualization in a Web Environment

Your task now is to create an **ASP.NET MVC Web Application** for visualizing a rating (a number from 0 to 100). 1 to 10 stars should be drawn (with halves). The stars should be generated with a **for** loop. The user interface might look like the following:



Creating a New C# Project

Create a new ASP.NET MVC web app with C# in Visual Studio. Add a new project from [Solution Explorer] -> [Add] -> [New Project...]. Give it a meaningful name, for example "WebApp-Ratings".



Choose the type of the app to be **MVC**.



Creating a View Holding a HTML Form

Open and edit the file `Views/Home/Index.cshtml`. Delete everything and insert the following code:

```

@{
    ViewBag.Title = "Rating";
}

<h1>Rating</h1>

<form action="DrawRating">
    <input type="number" min="0" max="100" name="rating" value="@ViewBag.Rating" />
    <input type="submit" value="Rate" />
</form>

<br />
@Html.Raw(ViewBag.Stars)

```

This code creates a web form `<form>` with a field `"rating"` for inputting a number in the range [0 ... 100] and a button `[Draw]` to send data from the form to the server. The action that will process the data is called `/Home/DrawRatings`, which means method `DrawRatings` in controller `Home`, which is in the file `HomeController.cs`. After the form the contents of `ViewBag.Stars` are printed. The code that will be inside will be dynamically generated by the HTML controller with a series of stars.

Adding the DrawRatings(int) Method

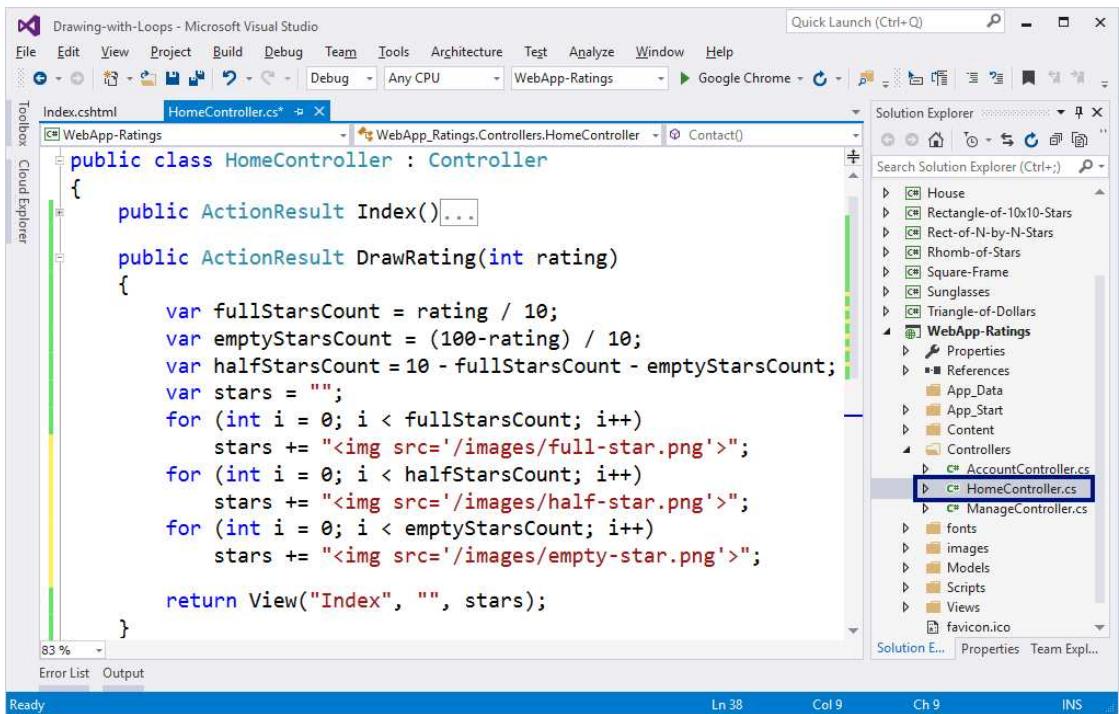
Add a method `DrawRatings` in the controller `HomeController`. Open the file `Controllers/HomeController.cs` and add the following code:

```
public ActionResult DrawRating(int rating)
{
    var fullStars = rating * 10 / 100;
    var emptyStars = (100 - rating) * 10 / 100;
    var halfStars = 10 - fullStars - emptyStars;

    var stars = "";
    for (int i = 0; i < fullStars; i++)
        stars += "<img src='/images/full-star.png' />";
    for (int i = 0; i < halfStars; i++)
        stars += "<img src='/images/half-star.png' />";
    for (int i = 0; i < emptyStars; i++)
        stars += "<img src='/images/empty-star.png' />";

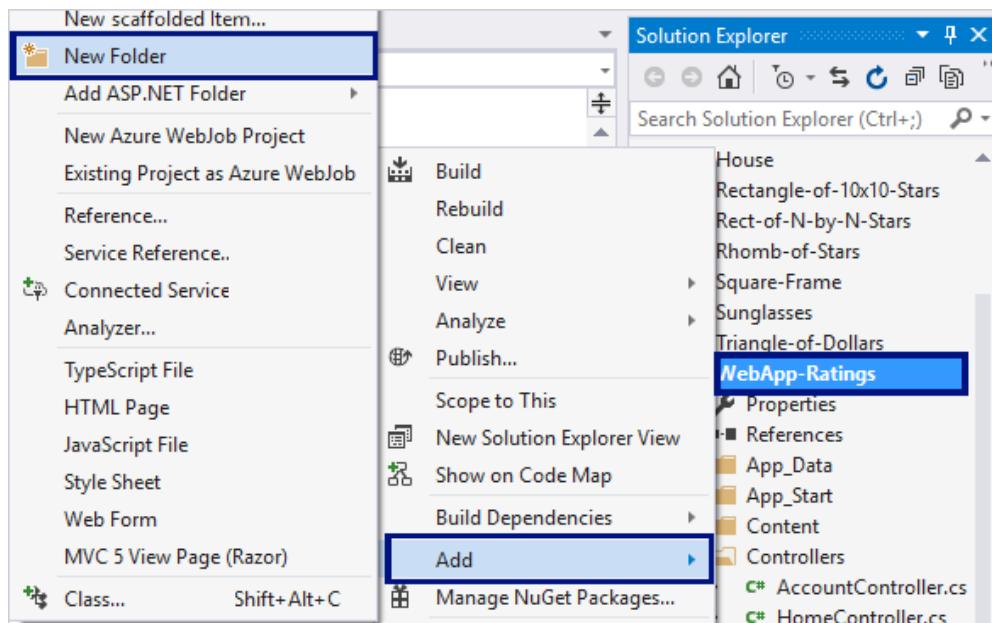
    ViewBag.Stars = stars;
    ViewBag.Rating = rating;
    return View("Index");
}
```

The above code takes the number `rating`, makes some calculations to find the number of **full stars**, the number of **empty stars** and the number of **half-full stars**, after which it generates an HTML code, which orders a few pictures of stars one after the other so that it can make the rating picture from them. The ready HTML code is stored in `ViewBag.Stars` to visualize the view `Index.cshtml`. Additionally, the sent rating is kept (as a number) in `ViewBag.Rating`, so that it can be put in the field for rating in the view. In order to ease your work, you can help yourself with the picture of Visual Studio below:

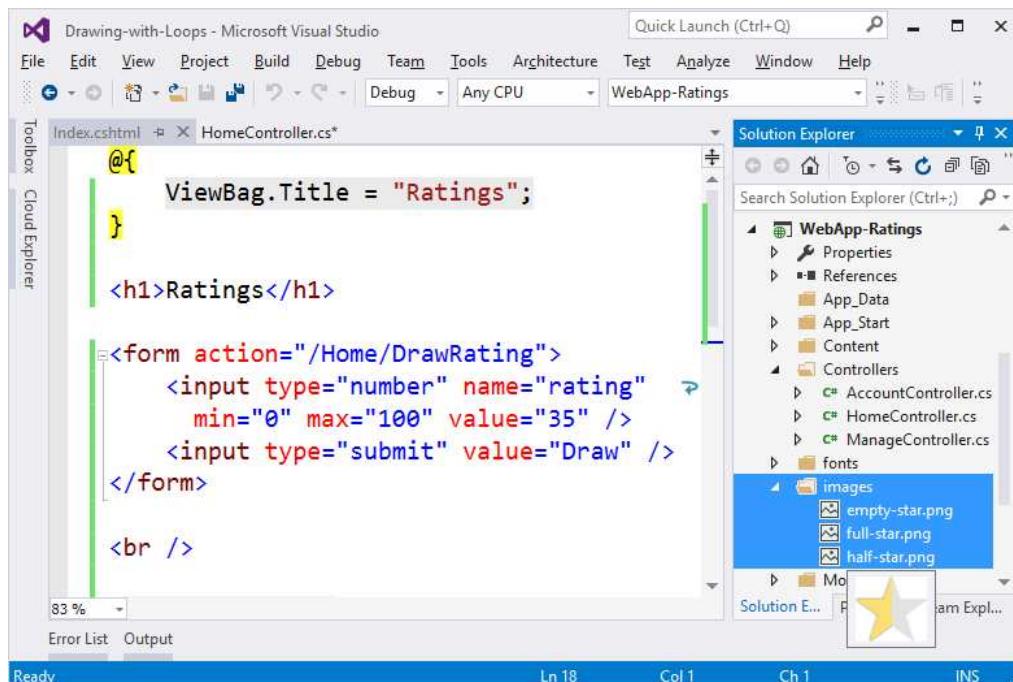


Adding Star Images

Create a new images folder in the project, using [Solution Explorer]:

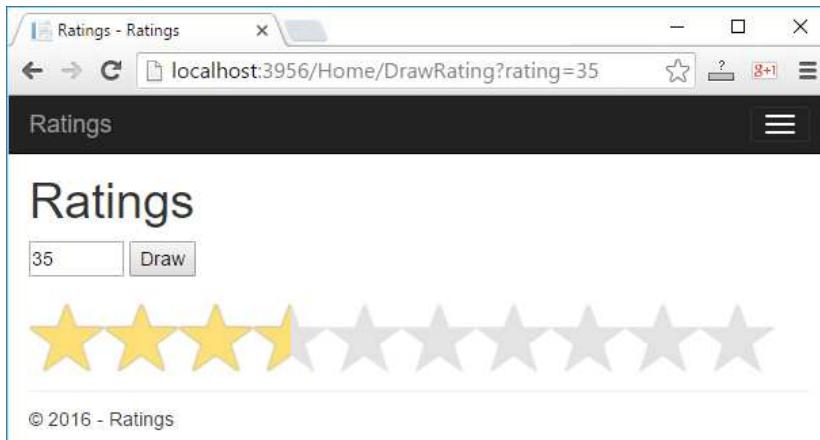


Now add **the pictures with the stars** (they are a part of the files with this project and can be downloaded from the book's GitHub repository: <https://github.com/SoftUni/Programming-Basics-Book-CSharp-EN/tree/master/assets/chapter-6-assets>). Copy them from Windows Explorer and paste them in the **images** folder in [Solution Explorer] in Visual Studio.



Starting and Testing the Project

Start the project with [Ctrl+F5] and enjoy:



If you have a problem with the example project above, you can ask in the SoftUni official **discussion forum** (<http://forum.softuni.org>) or in the SoftUni official **Facebook page** (<https://fb.com/softuni.org>).

High-quality education profession and job for software engineers

- ✓ Free start for **absolute beginners**
- ✓ Choose among the **most in-demand** professions in the software industry
- ✓ Assistance for **career start**
- ✓ **Top trainers**, mentors and active learning **community**



SoftUni provides practical, modern and **innovative education** for programming, IT, design, marketing and digital skills for thousands of young people. The "**Software University**" program builds true **software engineering professionals**.

Join the end-to-end **software engineering learning program** at SoftUni to master the most in-demand programming concepts, languages, software technologies and skills through a modern teaching methodology, **interactive learning platform** and tons of practical exercises and projects. Our **curriculum** is developed together with the **IT companies** to closely match the industry demands.

SoftUni works directly with the **companies** from the software industry to **find a job** for its students and to make them successful software engineers.

The path of the student at SoftUni



Apply now

softuni.org/apply

Chapter 6.2. Nested Loops – Exam Problems

In the previous chapter we introduced **nested loops** and how to use them to **draw** different kinds of figures on the **console**. Now we shall solve a few exam problems to gain more experience.

Nested Loops – Quick Review

We learned how to print figures with different sizes, thinking of an appropriate logic to construct them using **single** and **nested for** loops in combination with various calculations and program logic:

```
for (var r = 1; r <= 5; r++)
{
    Console.Write("*");
    for (var c = 1; c < 5; c++)
        Console.Write(" *");
    Console.WriteLine();
}
```

We also learned about the **new string constructor**, which lets you print a character a number of times defined by us:

```
string printMe = new string('*', 5);
```

Now, after the review, let's solve several **exam problems** related to nested loops to practice what we learned and to further develop our algorithmic thinking.

Problem: Drawing a Fort

Write a program that reads from the console **an integer n** and draws a **fort** $2 * n$ columns wide and **n rows** tall, as in the examples below. The left and right columns inside are $n / 2$ wide.

Input Data

The input is **an integer n** within the range [3 ... 1000].

Output Data

Print on the console **n** text rows, depicting **the fort** exactly as in the examples.

Sample Input and Output

Input	Output	Input	Output	Input	Output
3	/^\v/^\ _/_/_	4	/^\v\^\v\ _/_/_/_	5	/^\v__/\^\v\ _/_/_/_/_

Hints and Guidelines

Let's solve the problem step by step: read the input, perform some calculations, print the fort roof, print the fort body, print the fort base.

Reading the Input Data

We can see from the explanation that **the input data** will be only one line which will contain **an integer** within the range [3 ... 1000]. Therefore, we will use **a variable** of **int** type.

```
var n = int.Parse(Console.ReadLine());
```

After we have declared and initialized the input data, we must divide **the fort** into three parts:

- roof
- body
- base

Calculating and Printing the Roof

We can see from the examples that **the roof** is made of **two towers** and a **middle part**. Each tower has a beginning /, middle part ^ and an end \.



\ is a special symbol in C# and using only it in the method **Console.WriteLine(...)**, the console will not print it, that's why we show with \\ that we want to print exactly this symbol, without being interpreted as a special symbol (this is called "character escaping").

The size of the middle part is **n / 2**, therefore we can write this value in a new **variable**. It will keep **the size of the middle part of the tower**.

```
var colSize = n / 2;
```

Now we declare a second **variable**, in which will keep **the value** of the part **between the two towers**. The middle part of the roofs has a size of **2 * n - 2 * colSize - 4**.

```
var midSize = 2 * n - 2 * colSize - 4;
```

In order to print **the roof**, we will use **new string**, which takes two parameters (**char**, **int**) and connects a symbol **n** times.

```
Console.WriteLine("/{0}\\{1}/{0}\\",
    new string('^', colSize),
    new string('_', midSize));
```

Printing the Body of the Fort

The body of the fort contains a beginning |, a middle part (**white space**) and an end |. **The middle part** is a blank space with size of **2 * n - 2**. The number of **the rows** used for walls can be found in the given parameters: **n - 3**. This code prints the body of the fort:

```
for (var row = 1; row <= n - 3; row++)
{
    Console.WriteLine("|{0}|", new string(' ', 2 * n - 2));
}
```

Printing the Base of the Fort

In order to draw the last but one row, which is a part of the base, we need to print a beginning |, middle part (**white space**)_(**white space**) and an end |. In order to do this, we can use the

already declared variables `colSize` and `midSize` again, because we can see from the examples that they are equal to the `_` in the roof.

```
Console.WriteLine("|{0}{1}{0}|",
    new string(' ', colSize + 1),
    new string('_',         ));
```

We add `+ 1` to the white spaces, because we have **one** white space more in the examples.

The structure of **the base of the fort** is the same as the one of **the roof**. It is made of **two towers** and **a middle part**. Each **tower** begins with `\`, then a **middle part** `_` and an end `/`.

```
Console.WriteLine("\\"{0}"/{1}\\{0}/",
    new string('_',        ),
    new string(' ',       ));
```

Testing in the Judge System

Test your solution here: <https://judge.softuni.org/Contests/Practice/Index/513#0>.

Problem: Butterfly

Write a program that takes **an integer n** from the console and draws **a butterfly** $2 * n - 1$ columns wide and $2 * (n - 2) + 1$ rows tall, as in the examples below. **The left and right parts** are $n - 1$ wide.

Input Data

The input is **an integer n** within the range [3 ... 1000].

Output Data

Print on the console $2 * (n - 2) + 1$ text rows, depicting **the butterfly** as in the examples.

Sample Input and Output

Input	Output	Input	Output	Input	Output
3	*\ /* @ */ *	5	***\ /*** ---\ /--- ***\ /*** @ ***/ *** ---/ \--- ***/ ***	7	*****\ /***** -----\ /----- *****\ /***** -----\ /----- *****\ /***** @ *****/ ***** -----/ \----- *****/ ***** -----/ \----- *****/ *****

Hints and Guidelines

We can see in the explanation that **the input data** will be taken from only one row which contains **an integer** within the range [3 ... 1000]. This is why we will use **a variable** of **int** type.

```
var n = ...;
```

Divide the Figure into Parts

We can divide the figure into 3 parts – upper wing, body and lower wing. In order to draw the upper wing, we need to divide it into parts – beginning *, middle part \ / and end *. After looking at the examples we find out that the beginning is with size `n - 2`.

```
var halfRowSize = ...;
```

We can also see that the upper wing is with size `n - 2`, and that's why we can make a loop which repeats `halfRowSize` times.

```
for (int i = 1; i <= halfRowSize; i++)
{
}
```

We can see in the examples that on an even row we have a beginning *, a middle part \ / and an end *, and on an odd row – beginning -, middle part \ / and an end -. This is why we must add an **if-else** condition to check if the row is even or odd and then to draw one of the two types of rows.

```
for (int i = 1; i <= halfRowSize; i++)
{
    if (i % 2 != 0)
    {
        Console.WriteLine("{0} \\ / {0}",
            new string('*', ...));
    }
    else
    {
        ...
    }
}
```

Printing the Body and the Lower Wing

In order to create **the body of the butterfly** we can use **the variable `halfRowSize`** again and print on the console exactly **one** row. The body structure begins with **(white space)**, middle @ and ends with **(white space)**.

```
Console.WriteLine("{0} @ {0}",
    new string(' ', ...));
```

Now we need to print the lower wing, which is the same as the upper one.

```
for (int i = 1; i <= halfRowSize; i++)
{
    if (i % 2 != 0)
    {
        Console.WriteLine(
            new string(' ', ...));
    }
}
```

```

    }
    else
    {
        Console.WriteLine("STOP!");
    }
}

```

Testing in the Judge System

Test your solution here: <https://judge.softuni.org/Contests/Practice/Index/513#1>.

Problem: "Stop" Sign

Write a program that takes an integer **n** from the console and draws a warning sign STOP with size as in the examples below.

Input Data

The input is an integer **N** within the range [3 ... 1000].

Output Data

Print on the console text lines, which depict the warning sign STOP, as in the examples.

Sample Input and Output

Input	Output
3	<pre>_____//_____\ \ ...//_____\ \ ..//_____\ \ //__STOP!__\ __/ .__/ ..__/ </pre>

Input	Output
6	<pre>_____//_____\ \//_____\ \//_____\ \//_____\ \ ...//_____\ \ ..//_____\ \ ./__STOP!__\ __/ .__/ ..__/__/__/ </pre>

Hints and Guidelines

We can see from the explanation that the input data will come from only one line which contains an integer within the range [3 ... 1000]. Therefore, we will use a variable of **int** type.

```
int n = Console.ReadLine();
```

Divide the Figure into Parts

We can **divide** the figure into **3 parts** – upper, middle and lower. **The upper part** contains two sub-parts – first row and rows in which the sign widens. **The first row** is made of a beginning `.`, middle part `_` and an end `..`. After looking at the examples we can say that the beginning is `n + 1` columns wide, so it is good to write this **value** in a separate **variable**.

```
int dots = ...;
```

We must also create a second **variable**, in which we will keep the **value of the middle of the first row**, which has a size of `2 * n + 1`.

```
var underscores = 2 * n + 1;
```

After we have declared and initialized the two variables, we can print the first row on the console.

```
Console.WriteLine("{0}{1}{0}",
    ...,
    ...);
```

Printing the Upper Part of the Sign

In order to draw the rows in which the sign is getting "wider", we need to create a **loop**, which runs `n` times. The structure of a row contains a beginning `..`, `//` + middle part `_` + `\\"` and an end `..`. In order to reuse the already created **variables**, we need to decrease `dots` by 1 and `underscores` by 2, because we have already **printed** the first row, and the dots and underscores in the upper part of the figure are **decreasing**.

```
underscores -= 2;
dots--;
```

In each following iteration **the beginning** and **the end** decrease by 1, and the **middle part** increases by 2.

```
for (int i = 0; i < n; i++)
{
    Console.WriteLine("{0}//{1}\\\"{0}",
        new string('.', ...),
        new string('_', ...));

    underscores += 2;
    dots--;
}
```

Printing the Middle Row and the Lower Part

The **middle part** of the figure begins with `// + _`, middle part **STOP!** and an end `_ + \\\"`. The count of the underscores `_` is `(underscores - 5) / 2`.

```
Console.WriteLine("//{0}STOP!{0}\\\"", 
    new string('_', ...));
```

The **lower part** of the figure, in which the width of the sign **decreases**, can be done by creating another **loop**, which runs `n` times. The structure of a row is – a beginning `. + \\\"`, middle part `_` and an end `//`.

+ .. The number of **the dots** in the first iteration should be 0 and in each following one it **increases** by one. Therefore, we can say that the size of **the dots in the lower part of the figure** equals **i**.

```
for (int i = 0; i < n; i++)
{
    Console.WriteLine("{0}\\\\\\{1}//{0}",
        new string('.', i),
        new string('_', n - i));
}
```

In order for our program to work properly, in each iteration of **the loop** we need to **decrease** the number of **_** by **2**.

```
for (int i = 0; i < ; i++)
{
    // code here
    underscores -= ;
}
```

Testing in the Judge System

Test your solution here: <https://judge.softuni.org/Contests/Practice/Index/513#2>.

Problem: Arrow

Write a program that takes from the console **an integer n** and draws **a vertical arrow** with size as in the examples below.

Sample Input and Output

Input	Output	Input	Output
3	<pre>.###. .#.# ##.## .#.# ..#..</pre>	5	<pre>..#####.. ..#....#.. ..#....#.. ..#....#.. ####...### .#.....#. ..#....#..#....</pre>

Input Data

The input is **an odd integer n** within the range [3 ... 79].

Output Data

Print a vertical arrow on the console, in which "#" (number sign) marks the outline of the arrow, and "." – the rest.

Hints and Guidelines

From the explanation we see that **the input data** will be read from one input line only, which will contain **an integer** within the range [3 ... 1000]. This is why we will use a **variable** of **int** type.

```
int n = [REDACTED];
```

Divide the Figure into Parts

We can divide the figure into **3 parts** – upper, middle and lower one. **The upper part** contains two sub-parts – a first row and a body of the arrow. We can see from the examples that the count of **the outer dots** on the first row and in the body of the arrow is $(n - 1) / 2$. We can write this value in a **variable outerDots**.

```
var outerDots = (n - 1) / 2;
```

The number of **the inside dots** in the body of the arrow is $(n - 2)$. We must create a **variable** named **innerDots**, which will store this value.

```
var innerDots = n - 2;
```

We can see from the examples the structure of the first row. We must use the declared and initialized **variables outerDots** and **n**, in order to print **the first row**.

```
Console.WriteLine("{0}{1}{0}",
    new string('.', [REDACTED]),
    new string('#', [REDACTED]));
```

Printing the Body and the Middle Row

In order to draw **the body of the arrow**, we need to create **a loop**, which runs $n - 2$ times.

```
for (int i = 0; i < [REDACTED]; i++)
{
    Console.WriteLine("{0}#{1}#{0}",
        [REDACTED],
        [REDACTED]);
}
```

The middle of the figure is made of a beginning **#**, a middle **.** and an end **#**. The number of **#** is equal to **outerDots** and that's why we can use the same **variable** again.

```
Console.WriteLine("{0}{1}{0}",
    [REDACTED],
    [REDACTED]);
```

Printing the Lower Part of the Arrow

In order to draw **the lower part of the arrow**, we need to assign new values to **the variables outerDots** and **innerDots**.

```
outerDots = 1;
innerDots = 2 * n - 5;
```

Because `new string` can't join a symbol 0 times, the loop we are going to make must recur $n - 2$ times and we need to print the last row of the figure separately. At each iteration `outerDots` increases by 1, and `innerDots` decreases by 2.

```
for (int i = 0; i < n - 2; i++)
{
    Console.WriteLine(".....",
                      ".....",
                      ".....");
    outerDots++;
    innerDots -= 2;
}
```

The last figure row is made of a beginning `.`, a middle `#` and an end `..`. The number of `.` is `outerDots`.

```
Console.WriteLine(".....",
                  ".....",
                  ".....");
```

Testing in the Judge System

Test your solution here: <https://judge.softuni.org/Contests/Practice/Index/513#3>.

Problem: Axe

Write a program that takes an integer `n` and draws an axe with size as shown below. The width of the axe is $5 * N$ columns.

Sample Input and Output

Input
Output
2

```
-----**-----
-----*-*-
*****-*_
-----***-
```

Input
Output
5

```
-----**-----
-----*-*-
*****-*_
-----***-
-----**-----
-----*-*-
*****-*_
-----***-
-----**-----
-----*-*-
*****-*_
-----***-
```

Input
Output
8

```
-----**-----
-----*-*-
*****-*_
-----***-
-----**-----
-----*-*-
*****-*_
-----***-
-----**-----
-----*-*-
*****-*_
-----***-
-----**-----
-----*-*-
*****-*_
-----***-
-----**-----
-----*-*-
*****-*_
-----***-
```

Input Data

The input is an integer **n** in the range [2...42].

Output Data

Print an axe on the console as in the examples.

Hints and Guidelines

In order to solve the problem, we first need to calculate **the dashes in the left**, **the middle dashes**, **the dashes in the right** and the whole length of the figure.

```
width = 5 * n;
leftDashes = 3 * n;
middleDashes = 0;
rightDashes = width - leftDashes - middleDashes - 2;
```

Divide the Figure into Parts

We divide the figure into 3 parts: upper part, middle part (the handle), down part.

After we have declared and initialized **the variables**, we can begin drawing the figure by starting with **the upper part**. We can see from the examples what the structure of **the first row** is, and we can create a loop, which runs **n** times. At each iteration of the loop **the middle dashes** increase by 1, and **the right dashes** decrease by 1.

```
for (int i = 0; i < n; i++)
{
    Console.WriteLine("{0}*{1}*{2}",
        new string(' ', leftDashes),
        new string('#', middleDashes),
        new string(' ', rightDashes));
}
```

In order to use again **the variables** that we created in order to draw the handle of the axe, we need to decrease **the middle dashes** by 1, and we need to increase **the left dashes** by 1.

```
middleDashes++;
rightDashes--;
```

Printing the Handle

We can draw **the handle of the axe** by creating a loop, which runs **n - 2** times. We can see in the examples what its structure is.

```
(int i = 0; i < n - 2; i++)
{
    Console.WriteLine(" " + new string(' ', leftDashes) +
        new string('#', middleDashes) + " " + new string(' ', rightDashes));
}
```

```

    new string( ),
    new string( ),
    new string( ));
}

}

```

Printing the Lower Part of the Axe

We need to divide the lower part of the figure into two sub-parts – head of the axe and the last row of the figure. We will print the head of the axe on the console by creating a loop that runs $n / 2 - 1$ times. At each iteration the left dashes and the right dashes decrease by 1, and the middle dashes increase by 2.

```

for ( )
{
    Console.WriteLine("{0}*{1}*{2}",
        new string( ),
        new string( ),
        new string( ));

    // ...
}

}

```

For the last row of the figure we can use the three declared and initialized variables `leftDashes`, `middleDashes`, `rightDashes` again.

```

Console.WriteLine("",
    new string( ),
    new string( ),
    new string( ));

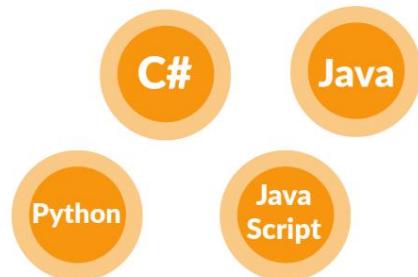
```

Testing in the Judge System

Test your solution here: <https://judge.softuni.org/Contests/Practice/Index/513#4>.

High-quality education profession and job for software engineers

- ✓ Free start for **absolute beginners**
- ✓ Choose among the **most in-demand** professions in the software industry
- ✓ Assistance for **career start**
- ✓ **Top trainers**, mentors and active learning **community**



SoftUni provides practical, modern and **innovative education** for programming, IT, design, marketing and digital skills for thousands of young people. The "**Software University**" program builds true **software engineering professionals**.

Join the end-to-end **software engineering learning program** at SoftUni to master the most in-demand programming concepts, languages, software technologies and skills through a modern teaching methodology, **interactive learning platform** and tons of practical exercises and projects. Our **curriculum** is developed together with the **IT companies** to closely match the industry demands.

SoftUni works directly with the **companies** from the software industry to **find a job** for its students and to make them successful software engineers.

The path of the student at SoftUni

[Apply now](#)softuni.org/apply

Chapter 7.1. More Complex Loops

Once we have learned what loops are and what the **for loops** serve for, now is the time to take a look at **other types of loops** as well as some **more complex loops constructions**. They will expand our knowledge and help us solve difficult and challenging problems. In particular, we will discuss how to use the following program constructions:

- loops with step
- **while** loops
- **do-while** loops
- **infinite** loops

In the current chapter, we will also learn how to exit from a loop using the **break** operator. Also, using the **try-catch** construction, we will learn how to handle **errors** during the program execution.

Video: Chapter Overview

Watch a video lesson to review what shall we learn in this chapter about the special and more complex types of loops: <https://youtu.be/J18RgaaMi7U>.

Introduction to More Complex Loops by Examples

Loops repeat a piece of code many times while a condition holds and usually changes the so called "loop variable" after each iteration. The loop variable using a **certain step**, e.g. 5 or -2. Example of a **for** loop from 10 down to 0, using a **step -2**:

```
for (int i = 10; i >= 0; i-=2)
    Console.WriteLine(i + " ");
// Output: 10 8 6 4 2 0
```

Run the above code example: <https://repl.it/@nakov/for-loop-step-minus-2-csharp>.

One of the simplest loops in programming is the **while-loop**. It repeats a block of code while a condition is true:

```
int n = 5;
int factorial = 1;
while (n > 1)
{
    factorial = factorial * n;
    n--;
}
Console.WriteLine(factorial);
// Output: 120
```

Run the above code example: <https://repl.it/@nakov/while-loop-factorial-csharp>.

Another example of loops is the **do-while loop**. It repeats a code block while a condition holds. For example, we can calculate the minimum number **k**, such that $2^k > 1,000,000,000$, using the code below:

```
int num = 1, count = 0;
do
{
```

```

    count++;
    num = num * 2;
} while (num <= 1000000000);

Console.WriteLine("2^{0} = {1}", count, num);
// Output: 2^30 = 1073741824

```

Run the above code example: <https://repl.it/@nakov/do-while-loop-power-of-2-csharp>.

Sometime in programming we don't know in advance how many times to repeat a loop, neither we have a clear loop condition, so we may use **infinite loop with exit condition** inside the loop. For example, we want to print the first 5 results, matching certain condition, calculated inside a loop. We use **infinite loop** and exit it using the **break operator**:

```

int value = 0, min = 100000, count = 0;
while (true)
{
    value = 2 * value + 1;
    if (value > min)
    {
        Console.WriteLine(value);
        count++;
    }

    if (count == 5)
        break;
}

```

Run the above code example: <https://repl.it/@nakov/infinite-loop-with-break-csharp>.

Let's get into details on how to use **for loops with a step**, how to use **while loops**, how to use **do-while loops** and how to design a program logic, based on **infinite loops with a break**.

For Loop with Step

In the "[Repetitions \(Loops\)](#)" chapter we learned how the **for** loop works and we already know when and for what purpose to use it. In this chapter we will pay **attention** to a particular and very important part of its construction, namely the **step**.

Video: Loop with a Step

Watch the following video lesson to learn how to use for-loops with a custom step: <https://youtu.be/QZDpWHcb7dE>.

Loop with a Step – Explanation

The **step** is that **part** of the **for** loop construction that tells **how** much to **increase** or **decrease** the value of its **leading** variable. It is declared the last in the skeleton of the **for** loop.

Most often, we have **a size of 1**, and in this case, instead of writing **i += 1** or **i -= 1**, we can use the **i++** or **i--** operators. If we want our step to be **different than 1**, when increasing, we use the **i += + step size** operator, and when decreasing, the **i -= + step size**. With step of 10, the loop would look like this:

```

int n = int.Parse(Console.ReadLine());
for (var i = 1; i <= n; i+=10)
{
    Console.WriteLine(i);
}

```

Setting a step

Here is a series of sample problems, the solution of which will help us better understand the use of the **step** in **for** loop.

Example: Numbers 1...N with Step 3

Write a program that prints the numbers from 1 to n with step of 3. For example, if n = 100, the result will be: 1, 4, 7, 10, ..., 94, 97, 100.

We can solve the problem using the following sequence of actions (algorithm):

- We read the number n from the console input.
- We run a **for loop** from 1 to n with step size of 3.
- In **the body of the loop**, we print the value of the current step.

```

int n = int.Parse(Console.ReadLine());

// by i+ =3 we increase the value of i with the size of the step
for (var i = 1; i <= n; i += 3)
{
    Console.WriteLine(i);
}

```

Testing in the Judge System

Test your solution here: <https://judge.softuni.org/Contests/Practice/Index/514#0>.

Example: Numbers N...1 in Reverse Order

Write a program that prints the numbers from n to 1 in reverse order (step of -1). For example, if n = 100, the result will be: 100, 99, 98, ..., 3, 2, 1.

Video: Numbers N...1

Watch this video lesson to learn how to print the numbers from N down to 1 (in reverse order) using a for loop: <https://youtu.be/LGPZ-ug3gh0>.

Hints and Guidelines

We can solve the problem in the following way:

- We read the number n from the console input.
- We create a **for loop** by assigning int i = n.
- We reverse the condition of the loop: i >= 1.
- We define the size of the step: -1.
- In **the body of the loop**, we print the value of the current step.

```
int n = int.Parse(Console.ReadLine());  
  
// Reversed condition: i >= 1  
// Negative step: i--  
for (int i = n; i >= 1; i--)  
{  
    Console.WriteLine(i);  
}
```

Testing in the Judge System

Test your solution here: <https://judge.softuni.org/Contests/Practice/Index/514#1>.

Example: Numbers from 1 to 2^n with a For Loop

In the following example, we will look at using the usual step with size of 1, combined with a calculation at each loop iteration.

Write a program that prints the numbers from 1 to 2^n (two in power of n). For example, if $n = 10$, the result will be: 1, 2, 4, 8, 16, 32, 64, 128, 256, 512, 1024.

Video: Numbers 1 ... 2^n

Watch this video lesson to learn how to iterate over the number from 1 to 2^n using a for-loop:
https://youtu.be/B2k_yx3EV0I.

Hints and Guidelines

The code below demonstrates how we can calculate the powers of 2 for given n using a for-loop with a calculation at the end of its body:

```
int n = int.Parse(Console.ReadLine());  
int num = 1;  
for (var i = 0; i <= n; i++)  
{  
    Console.WriteLine(num);  
    num = num * 2;  
}
```

Testing in the Judge System

Test your solution here: <https://judge.softuni.org/Contests/Practice/Index/514#2>.

Example: Even Powers of 2

Print the even powers of 2 to 2^n : $2^0, 2^2, 2^4, 2^6, 2^8, \dots, 2^n$. For example, if $n = 10$, the result will be: 1, 4, 16, 64, 256, 1024.

Video: Even Powers of 2

Watch this video lesson to learn how to print the even powers of 2 using a for loop with a step:
https://youtu.be/H8t4HGN_Ap4.

Hints and Guidelines

Here is how we can solve the problem using a for-loop with a step:

- We create a **num** variable for the current number to which we assign an initial **value of 1**.
- For a **step** of the loop, we set a value of **2**.
- In the **body of the loop**: we print the value of the current number and **increase the current number num 4 times** (according to the problem's description).

```
int n = int.Parse(Console.ReadLine());
int num = 1;
for (var i = 0; i <= n; i += 2)
{
    Console.WriteLine(num);
    num = num * 2 * 2;
}
```

Testing in the Judge System

Test your solution here: <https://judge.softuni.org/Contests/Practice/Index/514#3>.

While Loop

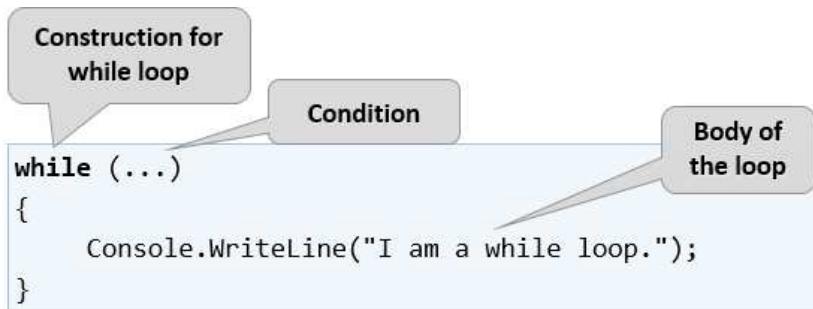
The next type of loops that we will get familiar with are called **while loops**. The specific thing about them is that they repeat a block of commands **while a condition is true**. As a structure, they differ from **for** loops, and even have a simple syntax.

Video: While Loop

Watch this video lesson to learn how to use the while-loop in C#: <https://youtu.be/Jqnxl6k1V9w>.

While Loop – Explanation

In programming the **while loop** is used when we want to **repeat** the execution of a certain logic while **a condition is in effect**. By "condition," we understand every **expression** that returns **true** or **false**. When the condition is **wrong**, the **while loop** is **interrupted**, the program **continues** to execute the remaining code after the loop. The **while loop** construction looks like this:



Here is a series of sample problems, the solution of which will help us better understand the use of the **while loop**.

Example: Sequence of Numbers $2k+1$

Write a program that prints all **numbers $\leq n$** of the series: **1, 3, 7, 15, 31, ...**, assuming that each next number = **previous number * 2 + 1**.

Here is how we can solve the problem:

- We create a **num** variable for the current number to which we assign an initial **value of 1**.
- For a loop condition, we put **the current number <= n**.
- In **the body of the loop**: we print the value of the current number and increase the current number by using the formula from the problem's description.

Here is a sample implementation of this idea:

```
int n = int.Parse(Console.ReadLine());
int num = 1;
while (num <= n)
{
    Console.WriteLine(num);
    num = 2 * num + 1;
}
```

Testing in the Judge System

Test your solution here: <https://judge.softuni.org/Contests/Practice/Index/514#4>.

Example: Number in Range [1...100]

Enter an integer in the range [1 ... 100]. If the entered number is **invalid**, enter it **again**. In this case, an invalid number will be any number that **is not** within the specified range.

Video: Numbers in the Range [1...100]

Watch this video lesson to learn how to enter a number in the range [1...100] using a while-loop:
<https://youtu.be/8W-CIbF4cdA>.

Hints and Guidelines

To solve the problem, we can use the following algorithm:

- We create a **num** variable to which we assign the integer value obtained from the console input.
- For a loop condition, we put an expression that is **true** if the number of the input **is not** in the range specified in the problem's description.
- In **the body of the loop**: we print a message "Invalid number!" on the console, then we assign a new value to **num** from the console input.
- Once we have validated the entered number, we print the value of the number outside the body of the loop.

Here's a sample implementation of the algorithm using a **while** loop:

```
var num = int.Parse(Console.ReadLine());
while (num < 1 || num > 100)
{
    Console.WriteLine("Invalid number!");
    num = int.Parse(Console.ReadLine());
}
Console.WriteLine("The number is: {0}", num);
```

Testing in the Judge System

Test your solution here: <https://judge.softuni.org/Contests/Practice/Index/514#5>.

Greatest Common Divisor (GCD)

Before proceeding to the next problem, we will get familiar with the definition of **the greatest common divisor (GCD)**, widely used in mathematics and numbers theory, and will learn how to calculate GCD.

Definition of GCD: the greatest common divisor of two **natural** numbers **a** and **b** is the largest number that divides **both a and b** without remainder.

a	b	GCD
24	16	8
67	18	1

a	b	GCD
12	24	12
15	9	3

a	b	GCD
10	10	10
100	88	4

Video: Greatest Common Divisor (GCD)

Watch the video lesson to learn about the Euclidean algorithm for calculating the GCD of given two integers: <https://youtu.be/1-SEOWupvrA>.

The Euclidean Algorithm

In the next problem we will use one of the first published algorithms for finding the GCD – **Euclid's algorithm**.

Until we reach a remainder of 0:

- We divide the greater number by the smaller one.
- We take the remainder of the division.

Euclid's algorithm **pseudo-code**:

```
while b ≠ 0
    var oldB = b;
    b = a % b;
    a = oldB;
print a;
```

Example: Greatest Common Divisor (GCD)

Enter **integers a and b** and find **GCD(a, b)**.

We will solve the problem through **Euclid's algorithm**:

- We create variables **a** and **b** to which we assign **integer** values taken from the console input.
- For a loop condition, we put an expression that is **true** if the number **b** is **different** from 0.
- In **the body of the loop** we follow the instructions from the pseudo code:
 - We create a temporary variable to which we assign **the current** value of **b**.
 - We assign a new value to **b**, which is the remainder of the division of **a** and **b**.
 - On the variable **a** we assign **the previous** value of the variable **b**.
- Once the loop is complete and we have found the GCD, we print it on the screen.

This is a sample **implementation** of the Euclidean algorithm:

```

var a = int.Parse(Console.ReadLine());
var b = int.Parse(Console.ReadLine());
while (b != 0)
{
    var oldB = b;
    b = a % b;
    a = oldB;
}
Console.WriteLine("GCD = {0}", a);

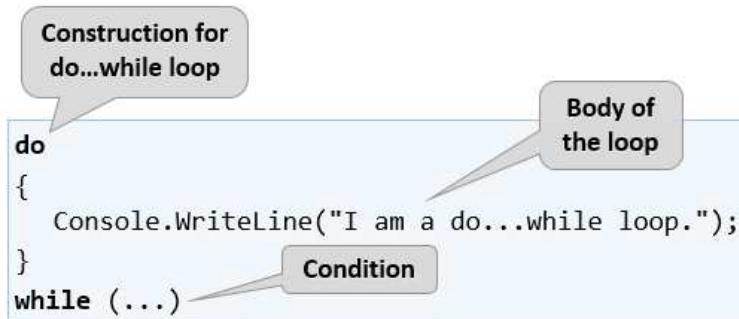
```

Testing in the Judge System

Test your solution here: <https://judge.softuni.org/Contests/Practice/Index/514#6>.

Do-While Loop

The next type of loops we will get familiar with are the **do-while** loops. By structure, this type of loop resembles the **while** loop, but there is a significant difference between them. It is that the **do-while** loop will execute its body **at least once**. Why is this happening? In the **do-while** loop construction, **the condition** is always checked **after** the body, which ensures that **the first loop iteration will execute the code and the check for the end of the loop** will be applied to each subsequent iteration of the **do-while**.



Now let's proceed to the usual set of sample practical problems follows. Their solutions will help us better understand the **do-while** loops.

Video: Do-While Loop

Watch a video lesson about the do-while loop and how to use it: <https://youtu.be/hEJ9-INyahU>.

Example: Calculating Factorial

For natural **n** number, calculate $n! = 1 * 2 * 3 * \dots * n$. For example, if $n = 5$, the result will be: $5! = 1 * 2 * 3 * 4 * 5 = 120$.

Here is how we can specifically calculate factorial:

- We create the variable **n** to which we assign an integer value taken from the console input.
- We create another variable – a **fact** which initial value is 1. We will use it for the calculation and storage of the factorial.
- For a loop condition, we will use **n > 1**, because each time we perform the calculations in the body of the loop, we will decrease the value of **n** by 1.

- In the body of the loop:
 - We assign a new value to **fact** that is the result of multiplying the current **fact** value to the current value of **n**.
 - We decrease the value of **n** by **-1**.
- Outside the body of the loop, we print the final factorial value.

This is a sample code, implementing the above described steps:

```
var n = int.Parse(Console.ReadLine());
var fact = 1;
do
{
    fact = fact * n;
    n--;
}
while (n > 1);
Console.WriteLine(fact);
```

Testing in the Judge System

Test your solution here: <https://judge.softuni.org/Contests/Practice/Index/514#7>.

Example: Summing Up Digits

Let's practice the **do-while** loop with the following exercise:

Sum up the digits of a **positive** integer **n**. Examples:

- If **n = 5634**, the result will be: $5 + 6 + 3 + 4 = 18$.
- If **n = 920**, the result will be: $9 + 2 + 0 = 11$.

Video: Sum of Digits

Watch this video lesson to learn how to sum the digits of given integer: <https://youtu.be/sbzIzdoEbFc>.

Hints and Guidelines

We can use the following **idea to solve the problem**: extract many times the last digit from the input number and sum the extracted digits until the input number reaches 0. Example:

- **sum = 0**
- **n = 5634** → extract 4; **sum += 4**; **n = 563**
- **n = 563** → extract 3; **sum += 3**; **n = 56**
- **n = 56** → extract 6; **sum += 6**; **n = 5**
- **n = 5** → extract 5; **sum += 5**; **n = 0** → end

In more detail the above idea looks like this:

- We create the variable **n**, to which we assign a value equal to the number entered by the user.
- We create a second variable – **sum**, which initial value is 0. We will use it for the calculation and storage of the result.
- As a loop condition, we will use **n > 0** because after each calculation of the result in the body of the loop, we will remove the last digit of **n**.

- In the body of the loop:
 - We assign a new value of **sum** that is the result of the sum of the current value of **sum** with the last digit of **n**.
 - We assign a new value to **n**, which is the result of removing the last digit of **n**.
- Outside the body of the loop, we print the final value of the sum.

This is a sample code, implementing the above described steps:

```
var n = int.Parse(Console.ReadLine());
var sum = 0;
do
{
    sum = sum + (n % 10);
    n = n / 10;
}
while (n > 0);
Console.WriteLine("Sum of digits: {0}", sum);
```



n % 10: returns the last digit of the number **n**.
n / 10: deletes the last digit of **n**.

Testing in the Judge System

Test your solution here: <https://judge.softuni.org/Contests/Practice/Index/514#8>.

Infinite Loops with Break

So far, we were introduced to various types of loops, learning what structures they have and how they are applied. Now, we need to understand what an **infinite loop** is, when it occurs, and how we can **break** it using the **break** operator.

Video: Infinite Loops with Break

Watch this video lesson to learn how to use infinite loops, along with the **break** operator: <https://youtu.be/rpez6b9TpA>.

Infinite Loop – Explanation

We call an infinite loop one that **repeats infinitely** the performance of its body. In **while** and **do-while** loops the end check is a conditional expression that **always** returns **true**. Infinite **for** occurs when there is **no condition to end the loop**.

Here is what an **infinite while** loop looks like:

```
while (true)
{
    Console.WriteLine("Infinite loop");
}
```

And here is what an **infinite for** loop looks like:

```
for (;;)
{
    Console.WriteLine("Infinite loop");
}
```

The Operator "Break"

We already know that the infinite loop performs a certain code infinitely, but what if we want at some point under a given condition to go out of the loop? The **break** operator comes in handy in this situation.



The **break** operator stops the execution of a loop at the time it is called and continues from the first line after the end of the loop. This means that the current iteration of the loop will not be completed, accordingly, the rest of the code in the body of the loop will not be executed.

Example: Prime Number Checking

The next problem we are going to solve is to **check whether given number is prime**. An integer is prime if it cannot be decomposed to a product of other numbers. For example: 2, 5 and 19 are primes, while 9, 12 and 35 are composite.

Video: Prime Number Checking

Watch this video lesson to learn how to design and implement an algorithm to check if given number is prime: <https://youtu.be/4lWOaPWkf0I>.

Hints and Guidelines

Before proceeding to the hints about solving the "prime checking" problem, let's recall in bigger detail what **prime numbers** are.

Definition: an integer is **prime** if it is **divisible only by itself and by 1**. By definition, the prime numbers are positive and greater than 1. The smallest prime number is 2.

We can assume that an integer **n** is a prime number if **n > 1** and **n** is not divisible by a number between 2 and **n-1**.

The first few prime numbers are: 2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, ...

Unlike the prime numbers, **composite numbers** are integers which can be obtained by multiplying several prime numbers.

Here are some examples of **composite numbers**:

- **10 = 2 * 5**
- **42 = 2 * 3 * 7**
- **143 = 13 * 11**

Positive integers, greater than 1, can be either **prime** or **composite** (product of primes). Numbers like **0** and **1** are not prime but are also not composite.

We can **check if an integer is prime** following the definition: check if **n > 1** and **n** is divisible by **2, 3, ..., n-1** without remainder.

- If it is divisible by any of the numbers, it is **composite**.
- If it is not divisible by any of the numbers, then it is **prime**.



We can optimize the algorithm instead of checking it to $n-1$, to check divisors to \sqrt{n} . Think what the reason for that is!

Prime Checking Algorithm

The most popular **algorithm** to check if a number n is prime is by checking if n is divisible by the numbers between 2 and \sqrt{n} .

The **steps** of the "prime checking algorithm" are given below in bigger detail:

- We create the variable **n** to which we assign an integer taken from the console input.
- We create an **isPrime** bool variable with an initial value **true**. We assume that a number is prime until proven otherwise.
- We create a **for** loop in which we set an initial value 2 for the loop variable, for condition **the current value $\leq \sqrt{n}$** . The loop step is 1.
- In **the body of the loop**, we check if **n**, divided by **the current value**, has a remainder. If there is **no remainder** from the division, then we change **isPrime** to **false** and we exit the loop through the **break** operator.
- Depending on the value of **isPrime**, we print whether the number is prime (**true**) or composite (**false**).

Implementation of the Prime Checking Algorithm

Here is a **sample implementation** of the prime checking algorithm, described above:

```
var n = int.Parse(Console.ReadLine());
var prime = true;
for (var i = 2; i <= Math.Sqrt(n); i++)
{
    if (n % i == 0)
    {
        prime = false;
        break;
    }
}
if (prime)
{
    Console.WriteLine("Prime");
}
else
{
    Console.WriteLine("Not prime");
}
```

What remains is to add a **condition** that checks if the input number is greater than **1**, because by definition numbers such as 0, 1, -1 and -2 are not prime.

Testing in the Judge System

Test your solution here: <https://judge.softuni.org/Contests/Practice/Index/514#9>.

Example: Enter an Even Number

The next example will be to write a program that **enters an even number** from the console. If an odd number is entered, the program should enter a number again, until an even number is entered.

We shall use an **infinite loop with break** to solve this problem, because we don't know how many times the loop body will be repeated.

We shall check if a particular number **n** is even, and if it is, we will print it on the screen. An even number is one that can be divided by 2 without remainder. If an invalid number is entered, we will ask the user to enter a number again and will display a notification that the input number is not even.

Hints and Guidelines

Here is an idea how we can implement the above described logic:

- We create a variable **n** to which we assign an initial value of **0**.
- We create an infinite **while** loop and as condition we will set **true**.
- In **the body of the loop**:
 - We take an integer value from the console input and assign it to **n**.
 - If **the number is even**, we exit the loop by **break**.
 - **Otherwise**, we display a message stating that **the number is not even**. The iterations continue until an even number is entered.
- Finally, after the loop, print the even number on the screen.

Implementation

```
var n = 0;
while (true)
{
    Console.WriteLine("Enter even number: ");
    n = int.Parse(Console.ReadLine());
    if (n % 2 == 0)
    {
        break; // even number -> exit from the loop
    }
    Console.WriteLine("The number is not even.");
}
Console.WriteLine("Even number entered: {0}", n);
```

Note: Although the code above is correct, it will not work if the user enters **text** instead of numbers, such as "Invalid number". Then parsing the text to a number will break and the program will display an **error message (exception)**. How to deal with this problem and how to capture and process exceptions using the **try-catch construction** will be learned later.

Testing in the Judge System

Test your solution here: <https://judge.softuni.org/Contests/Practice/Index/514#10>.

Nested Loops and Break

Once we have learned what **the nested loops** are and how the **break** operator works, it is time to figure out how they work together. For a better understanding, let's step by step write **a program** that

should make all possible combinations of **pairs of numbers**. The first number of the combination is increasing from 1 to 3 and the second one is decreasing from 3 to 1. The problem must continue running until **i + j** is not equal to 2 (**i = 1** and **j = 1**). The desired result is:

```
C:\WINDOWS\system32\cmd.exe
1 3
1 2
Press any key to continue . . .
```

Wrong Implementation

Here is a **wrong solution** that looks right at first sight:

```
for (int i = 1; i <= 3; i++)
{
    for (int j = 3; j >= 1; j--)
    {
        if (i + j == 2)
        {
            break;
        }
        Console.WriteLine(i + " " + j);
    }
}
```

If we leave our program that way, our result will be as follows:

```
C:\WINDOWS\system32\cmd.exe
1 3
1 2
2 3
2 2
2 1
3 3
3 2
3 1
Press any key to continue . . .
```

Why is it so? As we can see, the result is **missing "1 1"**. When the program reaches that point that **i = 1** and **j = 1**, it enters the **if** check and executes the **break** operation. This way, it **goes out of the inner loop**, but then continues the execution of the **outer loop**. **i** grows, the program enters the internal loop and prints the result.



When we use the **break** operator in a **nested loop**, it interrupts the execution of the inner loop **only**.

Correct Implementation

What is the **right solution**? One way to solve this problem is by declaring a **bool variable** to keep track if the loop rotation has to continue. If you need to exit (leave all nested loops), we set the variable to

true and exit the inner loop with a **break**, and in the next check we exit the outer loop. Here is an example implementation of this idea:

```
bool hasToEnd = false;
for (int i = 1; i <= 3; i++)
{
    if (hasToEnd == false)
    {
        for (int j = 3; j >= 1; j--)
        {
            if (i + j == 2)
            {
                hasToEnd = true;
                break;
            }
            Console.WriteLine(i + " " + j);
        }
    }
}
```

Thus, when **i + j = 2**, the program will set the **hasToEnd = true** and exit the inner loop. Upon the next rotation of the outer loop, through the check, the program will not be able to reach the inner loop and will interrupt its execution.

Testing in the Judge System

Test your solution here: <https://judge.softuni.org/Contests/Practice/Index/514#11>.

Handling Errors: Try-Catch

The last thing we will get familiar with in this chapter is how to "capture" **wrong data** using the **try-catch** construction.

Video: Using Try-Catch

Watch this video lesson to learn how to use the **try-catch** statement to enter a valid integer number in certain range: <https://youtu.be/OWLRjNcSh3I>.

What is Try-Catch?

The **try-catch** construction is used to **intercept and handle exceptions (errors)** during the program execution.

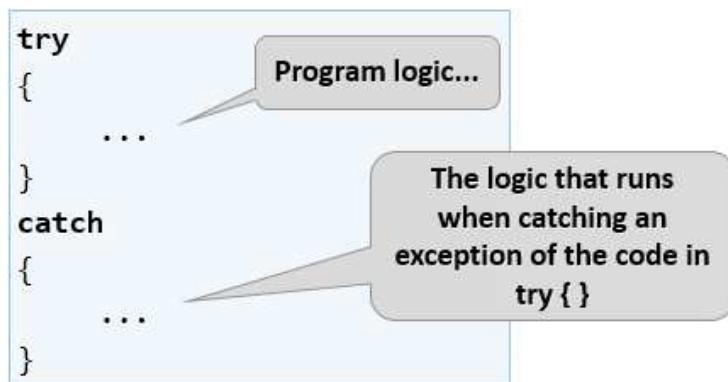
In programming, **exceptions** are a notification of an event that violates the normal operation of a program. Such exceptional events **interrupt the execution** of the program, and it is looking for something to process the situation. If it does not find it, the exception is printed on the console. If found, **the exception is processed**, and the program continues its normal execution. After a while, we'll see how this happens.

When an exception is found (e.g. when we divide an integer by zero), it is said that the exception was "thrown" (throw exception).

When the exception is **handled** and a piece of program logic recovers the program execution from the problem, we say the we "**catch the exception**".

The Try-Catch Construction

The **try-catch** construction in C# has different forms, but for now we will use the most basic of them:



We have a piece of code (sequence of commands) inside the **try** block. If this code **runs normally** (without errors), all the commands in the **try** blocks are executed. If some of the commands in the **try** block **throw an exception** (in case of an error), the code execution is stopped, and the **catch** block is executed. In this case we say that we **catch** and **handle** the error (exception).

In the next task, we will see how to handle a situation where a user enters a non-numeric input (for example, a **string** instead of an **int**) by **try-catch**.

Example: Dealing with Invalid Numbers with Try-Catch

Write a program that checks if an **n** number is even, and if it is, prints it on the screen. If an **invalid number is entered**, the program should display a notification that the entered input is not a valid number and the entering of the number has to be done again.

Here's how we can **solve the problem**:

- We create an infinite **while** loop and as a condition we set **true**.
- In the body of the loop:
 - We create a **try-catch** construction.
 - In the **try** block we write the programming logic for reading the user input, parsing it to a number, and the check for even number.
 - If it is **an even number**, we print it and go out of the loop (with **break**). The program is done and ends.
 - If it is **an odd number**, we print a message saying that an even number is required without leaving the loop (because we want it to be repeated again).
 - If we **catch an exception** when executing the **try** block, we write a message for invalid input number (and the loop is repeated because we do not explicitly go out of it).

Enter Even Number – Implementation

Here is a **sample implementation** of the described idea:

```

while (true)
{
    try
    {
        Console.Write("Enter even number: ");
        int n = int.Parse(Console.ReadLine());
        if (n % 2 == 0)
        {
            Console.WriteLine("Even number entered: {0}", n);
            break;
        }
        Console.WriteLine("The number is not even.");
    }
    catch
    {
        Console.WriteLine("Invalid number.");
    }
}

```

Play with the above code. Try to enter invalid numbers (e.g. text messages), non-integer numbers, odd numbers and even numbers.

The solution should **work in all cases**: whether we are entering integer numbers, invalid numbers (for example, too many digits), or non-numbered text.

The above program logic will repeat in an infinite loop the process of **entering a value until a valid even integer is entered**.

- The `int.Parse()` method will **throw an exception** in case of an invalid integer.
- In case of a valid integer, the program will check if it is even. In this case a "success" message is shown, and the **loop is stopped** using `break`.
- In case of an odd integer, an **error message** is shown, and the **loop repeats again**.
- In case of an exception (error during the number parsing), an **error message** is shown, and the **loop repeats again**.

Testing in the Judge System

Test your solution here: <https://judge.softuni.org/Contests/Practice/Index/514#12>.

Exercises: More Complex Loops

In this chapter, we got familiar with some new types of loops that can perform repetitions with more complex programming logic. Let's solve a few **practical problems** using these new constructs.

Video: Chapter Summary

Watch this video to review what we learned in this chapter: <https://youtu.be/6Wrna8QOLFA>.

What We Learned in This Chapter?

First, let's recall what we have learned.

We can use **for** loop with a step:

```
for (var i = 1; i <= n; i+=3)
{
    Console.WriteLine(i);
}
```

The **while** / **do-while** loops are repeated while a **condition** is true:

```
int num = 1;
while (num <= n)
{
    Console.WriteLine(num++);
}
```

If we have to **interrupt** the loop execution, we do it with the operator **break**:

```
var n = 0;
while (true)
{
    n = int.Parse(Console.ReadLine());
    if (n % 2 == 0)
    {
        break; // even number -> exit from the loop
    }
    Console.WriteLine("The number is not even.");
}
Console.WriteLine("Even number entered: {0}", n);
```

We can catch **errors** during the program execution:

```
try
{
    Console.Write("Enter even number: ");
    n = int.Parse(Console.ReadLine());
}
catch
{
    Console.WriteLine("Invalid number.");
}
// If int.Parse(...) fails, the catch { ... } block will execute
```

Now, let's work on a few exercises to practice the new loop types, learned from this chapter.

Problem: Fibonacci Numbers

Fibonacci's numbers in mathematics form a sequence that looks like this: 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89,

The formula to form the Fibonacci sequence is:

```
F0 = 1
F1 = 1
Fn = Fn-1 + Fn-2
```

Sample Input and Output

Input (n)	Output	Comment
10	89	$F(11) = F(9) + F(8)$
5	8	$F(5) = F(4) + F(3)$
20	10946	$F(20) = F(19) + F(18)$

Input (n)	Output
0	1
1	1
2	2

Enter an integer number n and calculate the n -number of Fibonacci.

Video: Fibonacci Numbers

Watch this video lesson to learn how to calculate the Fibonacci numbers using a **for** loop: <https://youtu.be/1ZROZBFzB3c>.

Hints and Guidelines

An idea to solve the problem:

- We create a **variable** n to which we assign an integer value from the console input.
- We create the variables $f0$ and $f1$ to which we assign a value of **1**, since the sequence starts.
- We create a **for** loop with condition **the current value $i < n - 1$** .
- In **the body of the loop**:
 - We create a **temporary** variable $fNext$, to which we assign the next number in the Fibonacci sequence.
 - To $f0$ we assign the current value of $f1$.
 - To $f1$ we assign the value of the temporary variable $fNext$.
- Out of the loop we print the n^{th} number of Fibonacci.

Example implementation:

```
var n = int.Parse(Console.ReadLine());
var f0 = 1;
var f1 = 1;
for (var i = 0; i < n - 1; i++)
{
    var fNext = f0 + f1;
    f0 = f1;
    f1 = fNext;
}
Console.WriteLine(f1);
```

Testing in the Judge System

Test your solution here: <https://judge.softuni.org/Contests/Practice/Index/514#13>.

Problem: Numbers Pyramid

Print **the numbers $1 \dots n$ in a pyramid** as in the examples below. On the first line we print one number, on the second line we print two numbers, on the third line we print three numbers, and so on, until the numbers are over. On the last line we print as many numbers as we get until we get to n .

Sample Input and Output

Input	Output	Input	Output	Input	Output	Input	Output
7	1 2 3 4 5 6 7	5	1 2 3 4 5	10	1 2 3 4 5 6 7 8 9 10	9	1 2 3 4 5 6 7 8 9

Video: Pyramid of Numbers

Watch this video lesson to learn how to draw a pyramid of numbers using nested loops and the **break** operator: <https://youtu.be/SWU-gQa31QI>.

Hints and Guidelines

We can solve the problem with **two nested loops** (by rows and columns) with printing in them and leaving when the last number is reached. Here is the idea, written in more details:

- We create a variable **n**, to which we assign an integer value from the console input.
- We create a variable **num** with an initial value of 1. It will keep the number of printed numbers. At each iteration we will **increase** it by 1 and print it.
- We create an **outer for** loop that will be responsible for the **rows** in the table. We name the variable of the loop **row** and set an initial value of 0. For condition, we set **row < n**. The size of the step is 1.
- In the body of the loop we create an **inner for** loop that will be responsible for the **columns** in the table. We name the variable of the loop **col** and set an initial value of 0. For a condition, we set **col < row** (**row** = number of digits per line). The size of the step is 1.
- In the body of the nested loop:
 - We check if **col > 1**, if yes -> we print space. If we do not do this, but directly print the space, we will have an unnecessary one at the beginning of each line.
 - We **print** the number **num** in the current cell of the table and **increase it by 1**.
 - We are checking for **num > n**. If **num** is greater than **n**, we **interrupt** the running of the **inner loop**.
- We print a **blank line** to move to the next one.
- Again, we check if **num > n**. If it is greater, we **interrupt our program by break**.

Implementation of the Idea

Here is a sample implementation:

```
var n = int.Parse(Console.ReadLine());
var num = 1;
for (var row = 1; row <= n; row++)
{
    for (var col = 1; col <= row; col++)
    {
        if (col > 1)
        {
            Console.Write(" ");
        }
        Console.Write(num);
        num++;
    }
    Console.WriteLine();
}
```

```

        Console.Write(" ");
    }
    Console.WriteLine();
    if (num > n)
    {
        break;
    }
}
Console.WriteLine();
if (num > n)
{
    break;
}
}

```

Testing in the Judge System

Test your solution here: <https://judge.softuni.org/Contests/Practice/Index/514#14>.

Problem: Numbers Table

Print the numbers 1 ... n in a table as in the examples below.

Sample Input and Output

Input	Output	Input	Output	Input	Output	Input	Output
3	1 2 3 2 3 2 3 2 1	4	1 2 3 4 2 3 4 3 3 4 3 2 4 3 2 1	2	1 2 2 1	5	1 2 3 4 5 2 3 4 5 4 3 4 5 4 3 4 5 4 3 2 5 4 3 2 1

Video: Table with Numbers

Watch this video lesson to learn how to print a table of numbers like the shown above using nested loops: <https://youtu.be/DVf7riptCwA>.

Hints and Guidelines

We can solve the problem using **two nested loops** and little calculations inside them:

- We read from the console the table size in an integer variable **n**.
- We create a **for** loop that will be responsible for the rows in the table. We name the loop variable **row** and assign it to an initial **value of 0**. As a condition, we set **row < n**. The step is 1.
- In **the body of the loop** we create a nested **for** loop that will be responsible for the columns in the table. We name the loop variable **col** and assign it an initial **value of 0**. As a condition, we set **col < n**. The size of the step is 1.
- In **the body of the nested loop**:

- We create a **num** variable to which we assign the result of **the current row + the current column + 1** (+1 as we start the count from 0).
- We check for **num > n**. If **num** is greater than **n**, we assign a new value to **num** which is equal to **two times n – the current value for num**. We do this in order not to exceed **n** in any of the cells in the table.
- We print the number from the current table cell.
- We print a **blank line** in the outer loop to move to the next row.

Implementation of the Idea

Here is a sample implementation of the described idea:

```
var n = int.Parse(Console.ReadLine());
for (int row = 0; row < n; row++)
{
    for (int col = 0; col < n; col++)
    {
        var num = row + col + 1;
        if (num > n)
        {
            num = 2 * n - num;
        }
        Console.Write(num + " ");
    }
    Console.WriteLine();
}
```

Testing in the Judge System

Test your solution here: <https://judge.softuni.org/Contests/Practice/Index/514#15>.

Lab: Web Application with Complex Loops

Now we know how to repeat a group of actions using **loops**. Let's do something interesting: a **web-based game**. Yes, a real game, with graphics and game logic. Let's have fun. It will be complicated, but if you do not understand how it works, relax. We are now entering into programming. You will advance with coding and with the software technologies over the time. For now, just follow the steps.

Problem: Web Application "Fruits Game"

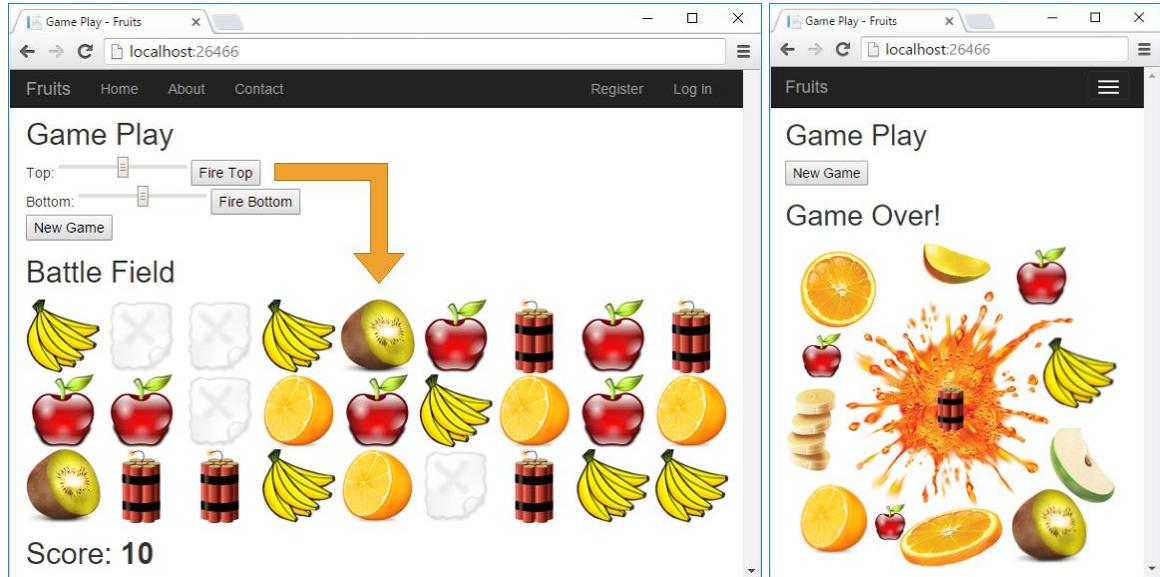
Description: Develop an **ASP.NET MVC Web Application** – a game in which the player **shoots fruits**, arranged in a table. Successfully hit fruits disappear and the player gets points for each target fruit. When you hit a **dynamite**, the fruits explode and the game ends (as in Fruit Ninja). The game is pretty simple, without animation and sound effects. It is designed to illustrate what apps you will be able to create later, when you learn C# and ASP.NET MVC web development.

Video: Fruits Game – ASP.NET MVC Web App

Watch this video lesson to learn how to build an ASP.NET MVC Web application "Fruits Game":
<https://youtu.be/inCr6SpHWC0>.

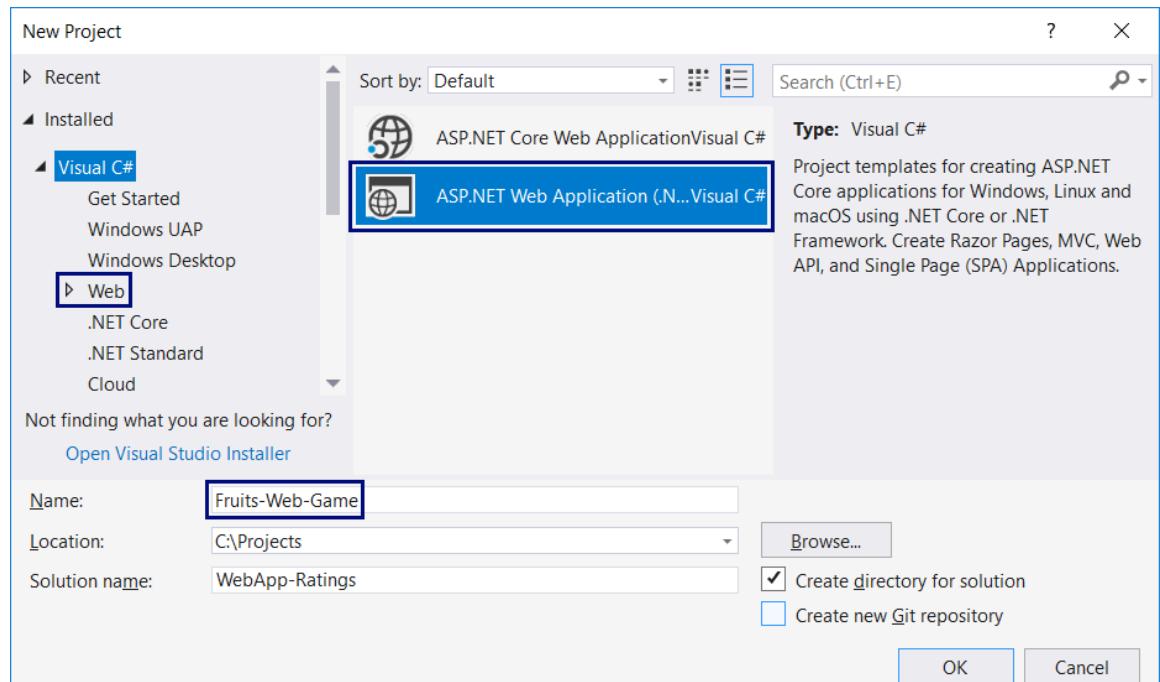
Fruits Game Explained

Shooting is done by columns, top to bottom or bottom to top, and the location of impact (the column under fire) is set by a scroll bar. Because of the inaccuracy of the scroller, the player is not quite sure which column they are going to shoot. Thus, every shot has a chance not to hit and this makes the game more interesting (like the sling in Angry Birds). Our game may look like this:



Create New C# Project

In Visual Studio, we create a new **ASP.NET MVC web application** with C# language. Add a new project from **[File] → [New] → [Project...]**. We give it a meaningful name, for example "**Fruits-Web-Game**".



Then we choose the type of Web app "MVC":



Create Controls

Now we will create the controls for the game. The goal is to add **scrolling bars** by which the player is targeting, and a button for starting a **new game**. We need to edit the file `Views/Home/Index.cshtml`. We delete everything in it and write the code from the picture:

```

@{
    ViewBag.Title = "Game Play";
}



## Game Play



<form action="/Home/FireTop">
    Top: <input type="range" name="position" min="0" max="100" />
    <input type="submit" value="Fire Top" />
</form>

<form action="/Home/FireBottom">
    Bottom: <input type="range" name="position" min="0" max="100" />
    <input type="submit" value="Fire Bottom" />
</form>

<form action="/Home/Reset">
    <input type="submit" value="New Game" />
</form>

```

This code creates an HTML form `<form>` with a scroller `position` for setting a number in the range [0 ... 100] and a button [Fire Top] for sending the form data to the server. The action that will process the data is called `Home/FireTop`, which means `FireTop` method in the `Home` controller, which is located in the file `HomeController.cs`. There are two similar forms with the [Fire Bottom] and [New Game] buttons.

Prepare Fruits for the View

Now we have to prepare the fruits for drawing in the view. Add the following code to the controller: `Controllers/HomeController.cs`:

```

Fruits-Web-Game - Microsoft Visual Studio
File Edit View Project Build Debug Team Tools Architecture Test Analyze Window Help
HomeController.cs > Index.cshtml
Fruits-Web-Game
public class HomeController : Controller
{
    static int rowsCount = 3;
    static int colsCount = 9;
    static string[,] fruits = GenerateRandomFruits();
    static int score = 0;
    static bool gameOver = false;

    public ActionResult Index()
    {
        ViewBag.rowsCount = rowsCount;
        ViewBag.colsCount = colsCount;
        ViewBag.fruits = fruits;
        ViewBag.score = score;
        ViewBag.gameOver = gameOver;
        return View();
    }
}

```

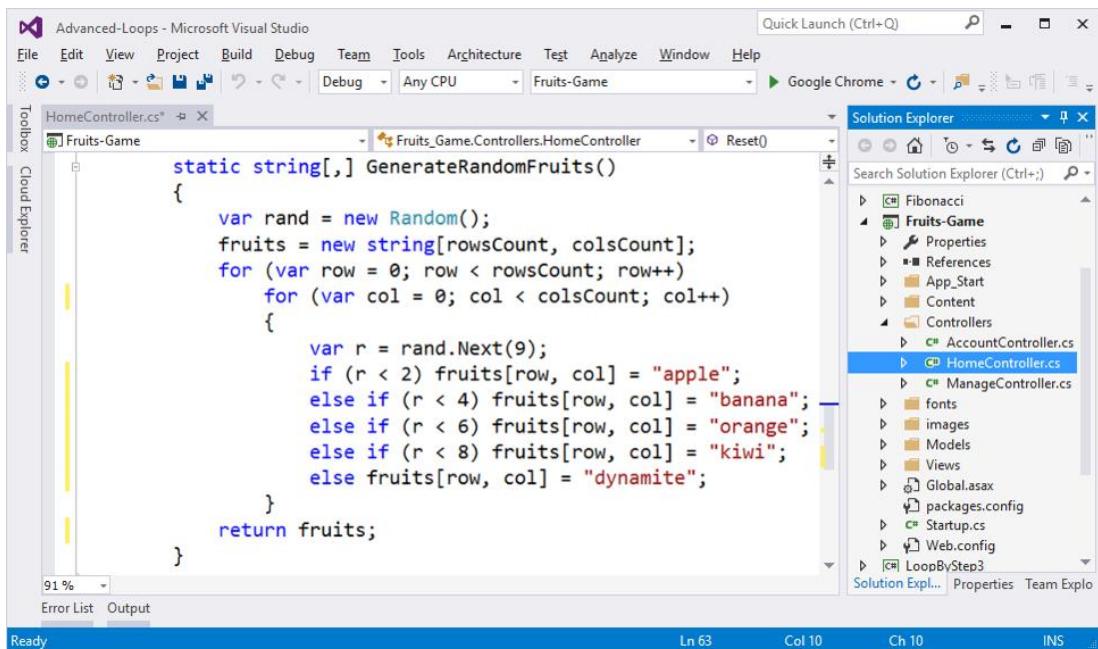
The screenshot shows the Microsoft Visual Studio interface. The main window displays the `HomeController.cs` file. The code defines a static matrix `fruits` containing random fruit names. The `Index()` action method populates a `ViewBag` with the number of rows, columns, the fruit matrix, the score, and the game over status. The Solution Explorer on the right shows the project structure, including files like `AccountController.cs`, `ManageController.cs`, and `Startup.cs`.

The above code defines the fields for **number of rows**, **number of columns**, **fruit table** (playing field), **points** accumulated by the player and information whether the game is active or **ended** (field `gameOver`). The playing field has 9 columns in 3 rows and contains for each field a text stating what is inside it: **apple**, **banana**, **orange**, **kiwi**, **empty** or **dynamite**. The main action `Index()` prepares the game field by recording in the `ViewBag` the structure of the game elements and invoking the view that draws them into the game page of the web browser as HTML.

Generate Random Fruits

We need to generate random fruits. To do this, we need to write a `GenerateRandomFruits()` method with the code from the image below. This code records in the matrix `fruits` names of different images and thus builds the playing field. Each cell of the table records one of the following values: **apple**, **banana**, **orange**, **kiwi**, **empty** or **dynamite**. Next, to draw the corresponding image in the view, the text of the table will be joined with the suffix `".png"` and this will give the name of the picture file that has to be inserted into the HTML page as part of the playing field. Filling in the playing field (9 columns with 3 rows) happens in the view `Index.cshtml` with two nested **for** loops (for row and column).

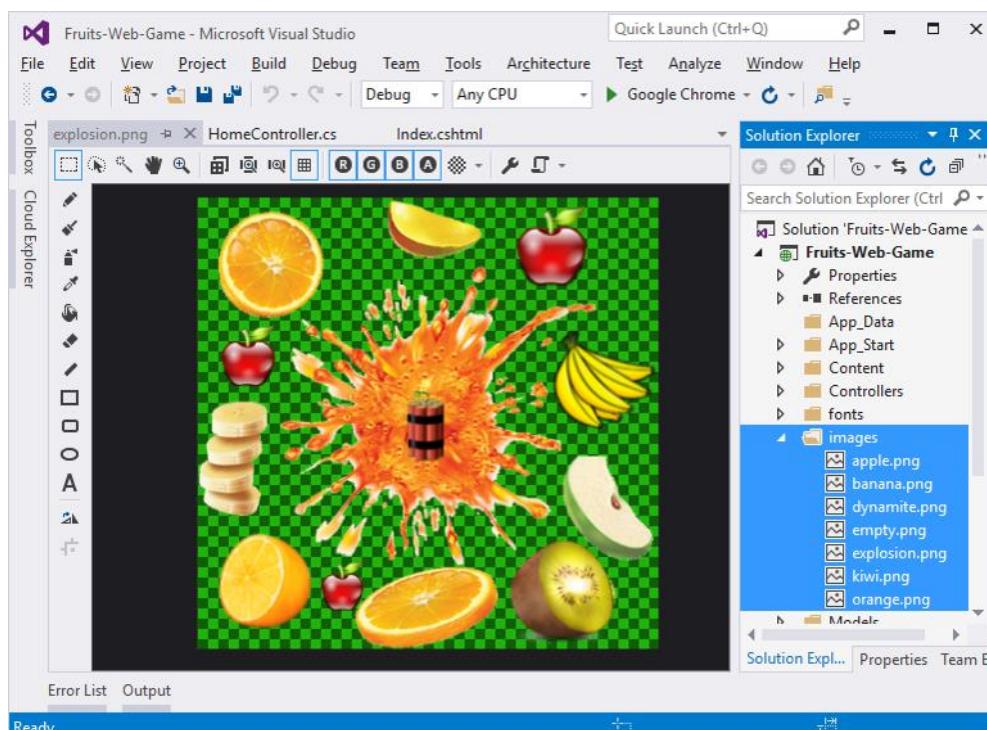
In order to generate random fruit for each cell, a **random number** is generated between 0 and 8 (see the class `Random` in .NET). If the number is 0 or 1, we place **apple**, if it is between 2 and 3, we place **banana** and so on. If the number is 8, we place **dynamite**. Obviously, the fruits appear twice as often as the dynamite. Here's the code:



Add Game Images

The next thing is to **add the images** for the game. From [Solution Explorer] create folder **images** in the root directory of the project. We use the menu **[Add] → [New Folder]**.

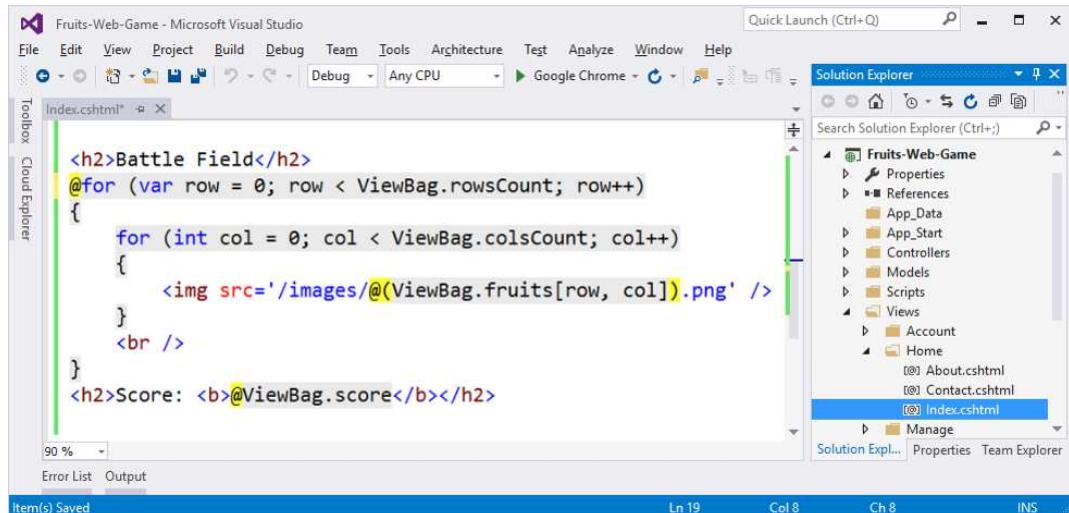
Now we add the **game images** (they are part of the project files for this project and can be downloaded from the book's GitHub repository: <https://github.com/SoftUni/Programming-Basics-Book-CSharp-EN/tree/master/assets/chapter-7-assets>). We copy them from Windows Explorer and put them in the **images** folder in [Solution Explorer] in Visual Studio with **copy/paste**.



Visualize Fruits

Drawing Fruits in `Index.cshtml`:

In order to **draw the playing field** with the fruits, we need to rotate two nested loops (for rows and columns). Each row consists of 9 images, each of which contains an **apple**, **banana** or other fruit, or empty **empty**, or **dynamite**. Images are drawn by printing an HTML tag to insert a picture: ``. Nine pictures are stacked one after the other on each row, followed by a new line with a `
`. This is repeated three times for the three lines. Finally, the player's points are printed. Here is what the code for drawing the playing field and points looks like:



```

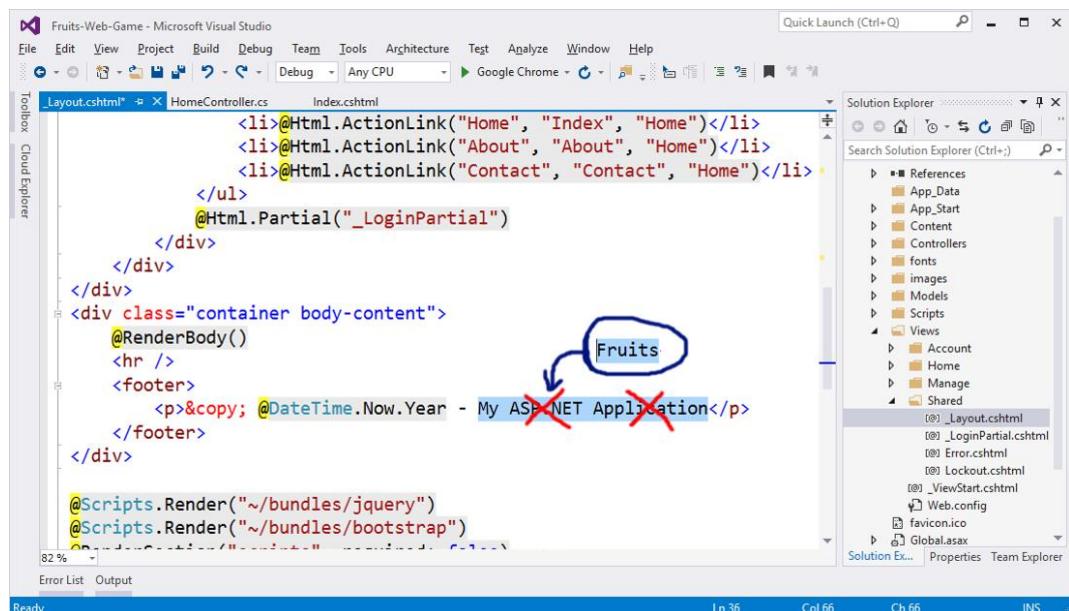
<h2>Battle Field</h2>
@for (var row = 0; row < ViewBag.rowsCount; row++)
{
    for (int col = 0; col < ViewBag.colsCount; col++)
    {
        <img src='@ViewBag.fruits[row, col].png' />
    }
    <br />
}
<h2>Score: <b>@ViewBag.score</b></h2>

```

Take a look at the yellow characters `@` – they are used to switch between the **C#** and **HTML** languages and come from the **Razor** syntax for drawing dynamic web pages.

Change Text in Layout

We need to adjust the texts in the `/Views/Shared/_Layout.cshtml` file. We replace **My ASP.NET Application** with more appropriate text, e.g. **Fruits**:



```

<ul>
    <li>@Html.ActionLink("Home", "Index", "Home")</li>
    <li>@Html.ActionLink("About", "About", "Home")</li>
    <li>@Html.ActionLink("Contact", "Contact", "Home")</li>
</ul>
@Html.Partial("_LoginPartial")

```

Fruits

```

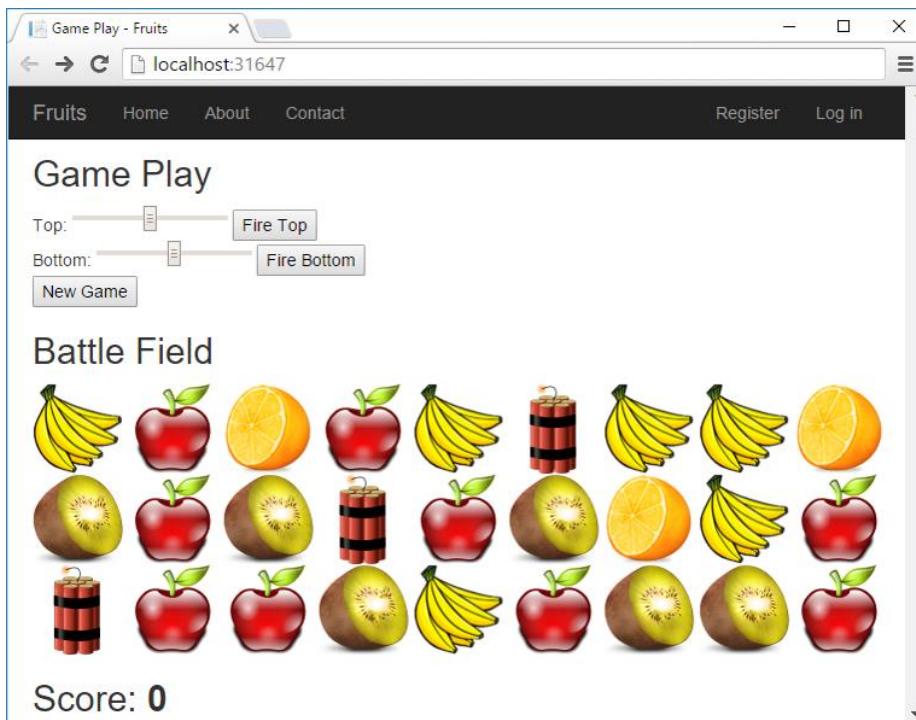
</div>
<div class="container body-content">
    @RenderBody()
    <hr />
    <footer>
        <p>&copy; @DateTime.Now.Year - Fruits</p>
    </footer>
</div>

@Scripts.Render("~/bundles/jquery")
@Scripts.Render("~/bundles/bootstrap")

```

Test the Application

Start the project using [Ctrl + F5] and enjoy it. It is expected to generate a random 9-to-3 playing field with fruits and visualize it on the web page through a series of pictures:



Now the game is sort of done: the playing field is randomly generated and rendered successfully (if you have not made a mistake somewhere). What remains is to fulfill the essence of the game: **shooting the fruits**.

Shooting the Fruits

For the fruit shooting, we need to add the actions [**Reset**] and [**Fire Top**] / [**Fire Bottom**] to the controller **HomeController.cs**:

```

public ActionResult Reset()
{
    fruits = GenerateRandomFruits();
    gameOver = false;
    return RedirectToAction("Index");
}

public ActionResult FireTop(int position)
{
    return Fire(position, 0, 1);
}

public ActionResult FireBottom(int position)
{
    return Fire(position, rowsCount - 1, -1);
}

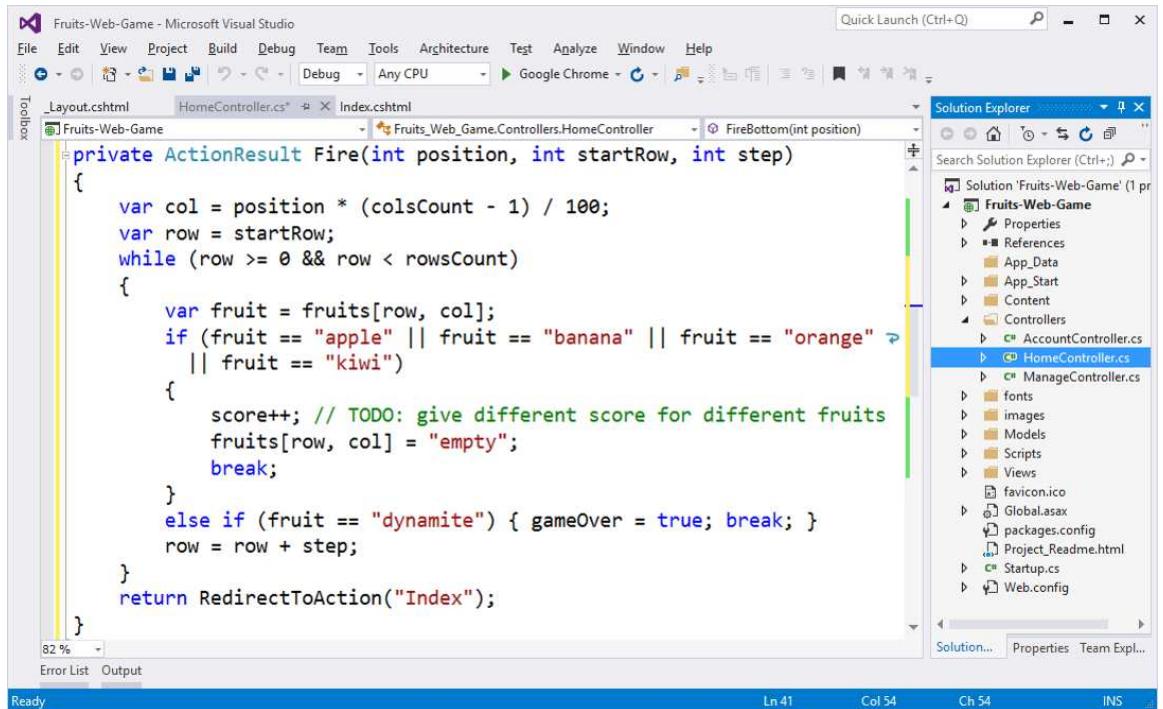
```

The above code defines three actions:

- **Reset()** – starts a new game by generating a new random playing field with fruits and explosives, resetting the player's points and making the game valid (`gameOver = false`). This action is pretty simple and can be immediately tested using [Ctrl + F5] before writing the others.
- **FireTop(position)** – shoots on row **0** at position **position** (number 0 to 100). The shooting is in direction **down** (+1) from row **0** (top). Shooting itself is more complicated as a logic and will be considered after a while.
- **FireBottom(position)** – shoots on row **2** at position **position** (number 0 to 100). The shooting is in direction **up** (-1) from row **2** (bottom).

Implement the "Fire" Method

We implement the "firing" method **Fire (position, startRow, step)**:



The screenshot shows the Microsoft Visual Studio interface with the following details:

- Title Bar:** Fruits-Web-Game - Microsoft Visual Studio
- Menu Bar:** File, Edit, View, Project, Build, Debug, Team, Tools, Architecture, Test, Analyze, Window, Help
- Toolbar:** Standard icons for file operations like Open, Save, Print, etc.
- Code Editor:** Displays the `HomeController.cs` file with the following code:

```
private ActionResult Fire(int position, int startRow, int step)
{
    var col = position * (colsCount - 1) / 100;
    var row = startRow;
    while (row >= 0 && row < rowsCount)
    {
        var fruit = fruits[row, col];
        if (fruit == "apple" || fruit == "banana" || fruit == "orange" ||
            || fruit == "kiwi")
        {
            score++; // TODO: give different score for different fruits
            fruits[row, col] = "empty";
            break;
        }
        else if (fruit == "dynamite") { gameOver = true; break; }
        row = row + step;
    }
    return RedirectToAction("Index");
}
```
- Solution Explorer:** Shows the project structure for 'Fruits-Web-Game' with files like AccountController.cs, HomeController.cs, ManageController.cs, etc.
- Status Bar:** Shows the current line (Ln 41), column (Col 54), character (Ch 54), and mode (INS).

Shooting works like this: first calculate the column number **col** to which the player has targeted. The input number from the scroll bar (between 0 and 100) is reduced to a number between 0 and 8 (for each of the 9 columns). Line number **row** is either 0 (if the shot is on top) or the number of lines minus one (if the shot is below). Accordingly, the shooting direction (step) is **1** (down) or **-1** (upwards).

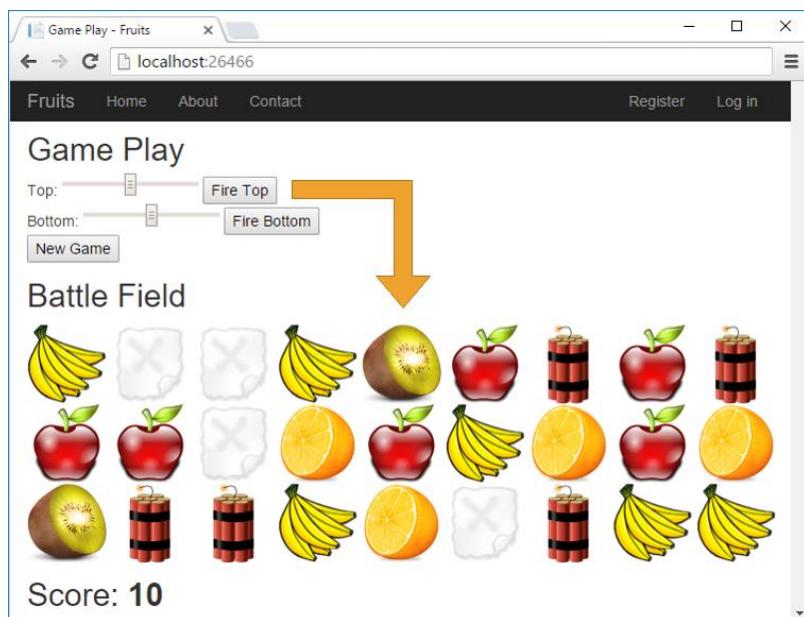
In order to find where the shot hits fruit or dynamite, go through a loop through all the cells in the playing field in the target column and from the first to the last attack row. If a fruit is hit, it disappears (replaced by **empty**) and points are given to the player. If the **dynamite** is hit, the game is marked as finished.

The more enthusiastic among you can implement a more complex behavior, for example, to give different points in the pursuit of a different fruit, to carry out animation with an explosion (this is not too easy), to take points in unnecessary firing in an empty column and so on.

Test the Application Again

We are testing what created up until now by starting with [Ctrl + F5]:

- **New Game** → the new game button aims to generate a new playing field holding randomly placed fruits and explosives and to reset the score of the player.
- **Shooting from top** → the top firing must remove the top fruit in the hit column or cause the game to end if there is dynamite. In fact, at the end of the game nothing is going to happen, because in the view this case is still not considered.
- **Shooting from bottom** → the shooting from bottom should remove the lowest fruit in the hit column or end the game when you hit the dynamite.



Implement "Game Over"

For now, at "End of the game" nothing happens. If a player reaches a dynamite, the controller says that the game is over (`gameOver = true`), but this fact is not visualized in any way. In order for the game to finish, we need to add several checks in the view:

```

@{
    ViewBag.Title = "Game Play";
}



## Game Play

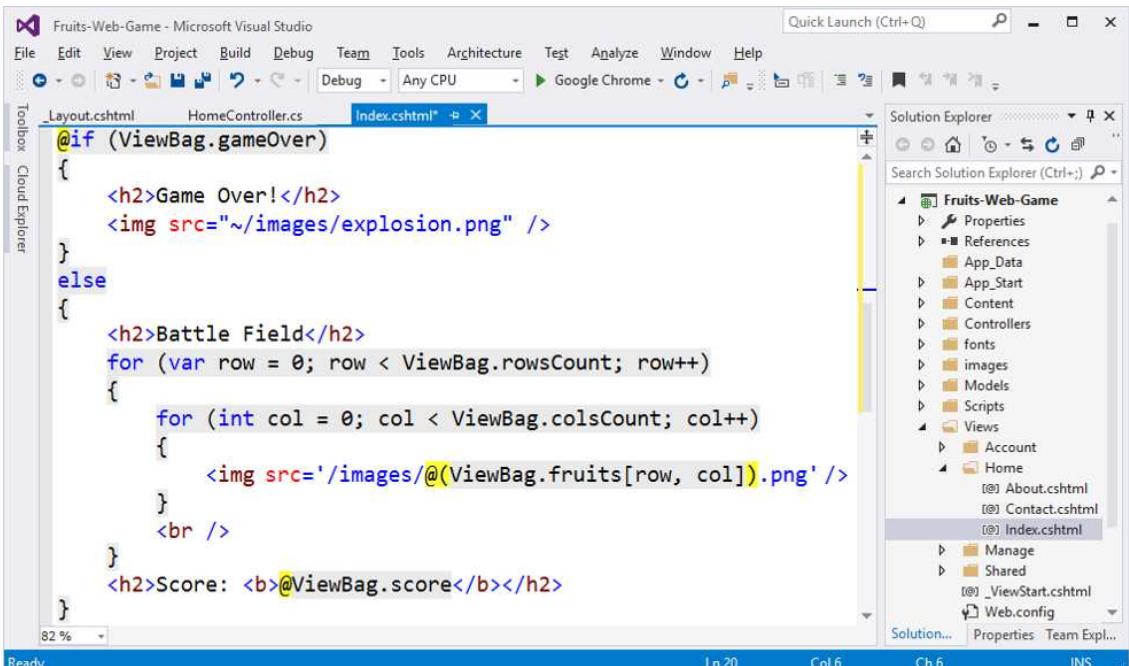


@if (!ViewBag.gameOver)
{
    <form action="/Home/FireTop">
        Top: <input type="range" name="position" min="0" max="100" />
        <input type="submit" value="Fire Top" />
    </form>
    <form action="/Home/FireBottom">
        Bottom: <input type="range" name="position" min="0" max="100" />
        <input type="submit" value="Fire Bottom" />
    </form>
}

<form action="/Home/Reset">
    <input type="submit" value="New Game" />
</form>

```

The new code in the view should look like this:



The screenshot shows the Microsoft Visual Studio interface with the 'Fruits-Web-Game' project open. The 'Index.cshtml' file is the active document. The code contains logic to check if the game is over and display the appropriate content based on the ViewBag variable.

```

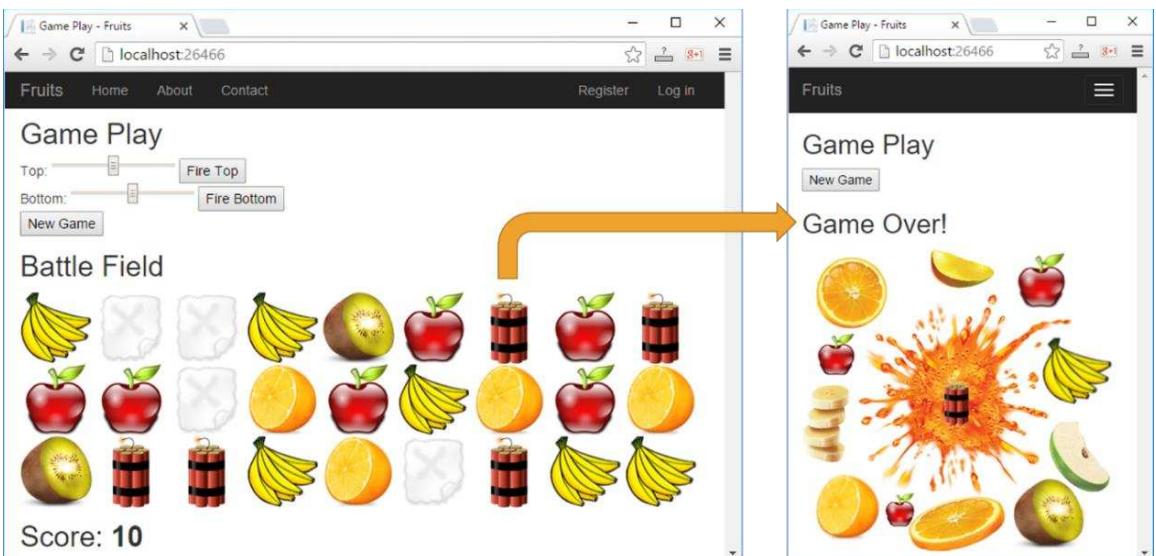
@if (ViewBag.gameOver)
{
    <h2>Game Over!</h2>
    
}
else
{
    <h2>Battle Field</h2>
    for (var row = 0; row < ViewBag.rowsCount; row++)
    {
        for (int col = 0; col < ViewBag.colsCount; col++)
        {
            <img src='/images/@(ViewBag.fruits[row, col]).png' />
        }
        <br />
    }
    <h2>Score: <b>@ViewBag.score</b></h2>
}

```

The code above checks whether the game has finished and indicates accordingly the shooting controls and the playing field (active game) or exploding fruit picture at the end of the game.

Final Testing of the Application

After changing the code in the view, let's start by [Ctrl + F5] and test the game again:



This time, when you hit a dynamite, the right picture should appear and allow only the "new game" action (the [New Game] button).

Was it complicated? Did you manage to create the game? If you have not succeeded, relax, this is a relatively complex project that includes a great deal of non-studied matter. If you want the Web game to pass through your hands, follow the above steps. In case of questions / problems, you ask for assistance in the SoftUni official discussion forum (<http://forum.softuni.org>) or in the SoftUni official Facebook page (<https://fb.com/softuni.org>).

High-quality education profession and job for software engineers

- ✓ Free start for **absolute beginners**
- ✓ Choose among the **most in-demand** professions in the software industry
- ✓ Assistance for **career start**
- ✓ **Top trainers**, mentors and active learning **community**



SoftUni provides practical, modern and **innovative education** for programming, IT, design, marketing and digital skills for thousands of young people. The "**Software University**" program builds true **software engineering professionals**.

Join the end-to-end **software engineering learning program** at SoftUni to master the most in-demand programming concepts, languages, software technologies and skills through a modern teaching methodology, **interactive learning platform** and tons of practical exercises and projects. Our **curriculum** is developed together with the **IT companies** to closely match the industry demands.

SoftUni works directly with the **companies** from the software industry to **find a job** for its students and to make them successful software engineers.

The path of the student at SoftUni



Apply now

softuni.org/apply

Chapter 7.2. More Complex Loops – Exam Problems

In the previous chapter we learned **how to execute** a block of commands **more than once** using a **for** loop. To improve our knowledge and skills, let's solve some **more complicated problems with loops**, given at the exams in SoftUni.

More Complex Loops – Quick Review

In the previous chapter **we reviewed some loop structures** that would help us solve more complex problems:

- **loops with a step** (e.g. print the numbers 1, 3, 5, ..., n)
- **nested** loops (loops located inside other loops)
- **while** loops (repeat a block of code while an entrance condition is true)
- **do-while** loop (repeat a block of code while an exit condition is true)
- **infinite** loops and breaking out of loop (**break** operator)
- the **try-catch** construction (handle runtime errors)

Problem: Dumb Passwords Generator

Write a program that enters two integers **n** and **l** and generates in alphabetical order all possible "dumb" **passwords** that consist of the following **5 characters**:

- Character 1: digit from **1** to **n**.
- Character 2: digit from **1** to **n**.
- Character 3: small letter among the first **l** letters of the Latin alphabet.
- Character 4: small letter among the first **l** letters of the Latin alphabet.
- Character 5: digit from **1** to **n**, **bigger than first 2 digits**.

Sample Input and Output

Input	Output
2	11aa2 11ab2 11ac2 11ad2 11ba2 11bb2 11bc2 11bd2
4	11ca2 11cb2 11cc2 11cd2 11da2 11db2 11dc2 11dd2

Input	Output
3	11aa2 11aa3 12aa3
1	21aa3 22aa3

Input	Output
4	11aa2 11aa3 11aa4 11ab2 11ab3 11ab4 11ba2 11ba3 11ba4 11bb2 11bb3 11bb4 12aa3 12aa4 12ab3 12ab4 12ba3 12ba4 12bb3 12bb4 13aa4 13ab4 13ba4 13bb4 21aa3 21aa4 21ab3 21ab4 21ba3 21ba4 21bb3 21bb4 22aa3 22aa4 22ab3 22ab4 22ba3 22ba4 22bb3 22bb4 23aa4 23ab4 23ba4 23bb4 31aa4 31ab4 31ba4 31bb4 32aa4 32ab4 32ba4 32bb4 33aa4 33ab4 33ba4 33bb4
2	

Input	Output
3	11aa2 11aa3 11ab2 11ab3 11ba2 11ba3 11bb2 11bb3 12aa3
2	12ab3 12ba3 12bb3 21aa3 21ab3 21ba3 21bb3 22aa3 22ab3 22ba3 22bb3

Input Data

The input is read from the console and consists of **two integers: n** and **l** within the range [1 ... 9].

Output Data

Print on the console all "dumb" passwords in alphabetical order, separated by space.

Hints and Guidelines

We can split the solution of the problem into 3 parts:

- **Reading the input** – in the current problem this includes reading two numbers **n** and **l**, each on a single line.
- **Processing the input data** – using of nested loops to iterate through every possible symbol for each of the five password symbols.
- **Printing the output** – printing every "dumb" password that meets the requirements.

Reading the Input Data

For reading of **input** data we will declare two integer variables **int: n** and **l**.

```
var n = int.Parse(Console.ReadLine());  
var l = int.Parse(Console.ReadLine());
```

Let's declare and initialize the **variables** that will store the **characters** of the password: for the characters of **digit** type – **int** – **d1, d2, d3**, and for the **letters** – of **char** type – **l1, l2**. To make it easier we will skip explicit specification of the type by replacing it with the keyword **var**.

Processing the Input Data and Printing Output

We need to create **five for** nested loops, one for each variable. To ensure that the last digit **d3** is **greater** than the first two, we will use the built-in function **Math.Max(...)**.

```
for (var d1 = 1; d1 <= n; d1++)  
{  
    for (var d2 = 1; d2 <= n; d2++)  
    {  
        for (var l1 = 'a'; l1 < 'a' + 1; l1++)  
        {  
            for (var l2 = 'a'; l2 < 'a' + 1; l2++)  
            {  
                for (var d3 = Math.Max(d1, d2) + 1; d3 <= n; d3++)  
                {  
                    Console.Write("{0}{1}{2}{3}{4} ",  
                        d1, d2, l1, l2, d3);  
                }  
                Console.WriteLine();  
            }  
        }  
    }  
}
```

Testing in the Judge System

Test your solution here: <https://judge.softuni.org/Contests/Practice/Index/515#0>.

Did you Know That...?

- In C# we can define a `for` loop with variable of `char` type:

```
for (char ch = 'a'; ch < 'z'; ch++)
```

- We can `read` variable of `char` type from the console with the following structure:

```
char ch = (char)Console.Read();
```

- We can convert a `Capital` char into `small` one, using a built-in C# function:

```
Char.ToLower(ch);
```

- By reading the chars from the console, we can directly convert upper to lowercase letters, by combining the above two lines:

```
char ch = Char.ToLower((char)Console.Read());
```

Problem: Magic Numbers

Write a program that enters a single integer `magic` number and produces all possible **6-digit numbers** for which the product of their digits is equal to the magical number.

Example: "Magic number" → 2

- 111112 → $1 * 1 * 1 * 1 * 1 * 2 = 2$
- 111121 → $1 * 1 * 1 * 1 * 2 * 1 = 2$
- 111211 → $1 * 1 * 1 * 2 * 1 * 1 = 2$
- 112111 → $1 * 1 * 2 * 1 * 1 * 1 = 2$
- 121111 → $1 * 2 * 1 * 1 * 1 * 1 = 2$
- 211111 → $2 * 1 * 1 * 1 * 1 * 1 = 2$

Input Data

The input is read from the console and consists of **one integer** within the range [1 ... 600 000].

Output Data

Print on the console **all magic numbers**, separated by **space**.

Sample Input and Output

Input	2
Output	111112 111121 111211 112111 121111 211111
Input	8
Output	111118 111124 111142 111181 111214 111222 111241 111412 111421 111811 112114 112122 112141 112212 112221 112411 114112 114121 114211 118111 121114 121122 121141 121212 121221 121411 122112 122121 122211 124111 141112 141121 141211 142111 181111 211114 211122 211141 211212 211221 211411 212112 212121 212211 214111 221112 221121 221211 222111 241111 411112 411121 411211 412111 421111 811111
Input	531441
Output	999999

Solution using a "For" Loop

The solution follows **the same** concept (again we need to generate all combinations for the n element). Following these steps, try to solve the problem yourself.

- Declare and initialize **variable** of **int** type and read the **input** from the console.
- Nest **six for loops** one into another, for every single digit of the searched 6-digit numbers.
- In the last loop, using **if**, check if the **product** of the six digits is **equal** to the **magic number**.

```
int magic = ...;
for (int d1 = 0; d1 <= 9; d1++)
{
    for (int d2 = 0; ... )
    {
        for ( ... )
        {
            for ( ... )
            {
                for ( ... )
                {
                    for ( ... )
                    {
                        if (d1 * d2 * d3 * d4 * d5 * d6)
                        {
                            ...
                        }
                    }
                }
            }
        }
    }
}
```

Solution using a "While" Loop

In the previous chapter we reviewed other loop constructions. Let's look at the sample solution of the same problem using the **while** loop.

First, we need to store the **input magical number** in a suitable variable. Then we will initialize 6 variables – one for each of the six digits of the **searched numbers**.

```
int magic = int.Parse(Console.ReadLine());
int d1 = 0;
int d2;
int d3;
int d4;
int d5;
int d6;
```

Writing a While Loop

Then we will start writing **while** loops.

- We will initialize **first digit: d1 = 0**.
- We will set a **condition for each loop**: the digit will be less than or equal to 9.
- In the **beginning** of each loop we set a value of the **next digit**, in this case: **d2 = 0**. In the nested **for** loops we initialize the variables in the inner loops at each increment of the outer ones. We want to do the same here.
- At **the end** of each loop we will increase the digit by one: **d++**.
- In the **innermost** loop we will make **the check** and if necessary, we will print on the console.

```
while (d1 <= 9)
{
    d2 = 0;
    while (d3 <= 9)
    {
        d3 = 0;
        while (d3 <= 9)
        {
            if (d1 * d2 * d3 * d4 * d5 * d6)
            {
                Console.WriteLine("{0}{1}{2}{3}{4}{5} ",
                    d1, d2, d3, d4, d5, d6);

                d3++;
            }
            d2++;
        }
        d1++;
    }
}
```

Infinite While Loop

Let's remove the **if** check from the innermost loop. Now, let's initialize each variable outside of the loops and delete the rows **dx = 0**. After we run the program, we only get 10 results. Why? What if you use **do-while**? In this case this loop does not look appropriate, does it? Think why. Of course, you can solve the problem using an **infinite loop**.

```
int d1 = 0;
while (true)
{
    int d2 = 0;
    while (true)
    {
        Console.WriteLine("{0}{1}", d1, d2);
        d2++;
        if (d2 == 10)
        {
            break;
        }
    }
    d1++;
    if (d1 == 10)
    {
        break;
    }
}
```

As we can see, we can solve a problem using different types of loops. Of course, each task has its most appropriate choice. In order to practice each type of loops – try to solve each of the following problems with all the learned loops.

Testing in the Judge System

Test your solution here: <https://judge.softuni.org/Contests/Practice/Index/515#1>.

Problem: Stop Number

Write a program that prints on the console all numbers from **N** to **M**, that are **divisible by 2 and 3 without remainder**, in **reversed order**. We will read one more "stop" number from the console – **S**. If any of the numbers divisible by 2 and 3 **is equal to the stop number**, **it should not be printed**, and the program should end. **Otherwise print all numbers up to N**, that meet the condition.

Input Data

Read from the console 3 numbers, each on a single line:

- **N** – integer number: $0 \leq N < M$.
- **M** – integer number: $N < M \leq 10000$.
- **S** – integer number: $N \leq S \leq M$.

Output Data

Print on the console on a single line all numbers, that meet the condition, separated by space.

Sample Input and Output

Input	Output	Comments
1 30 15	30 24 18 12 6	Numbers from 30 to 1, that are divisible at the same time by 2 and 3 without remainder are: 30, 24, 18, 12 and 6. The number 15 is not equal to any, so the sequence continues.

Input	Output	Comments
1 36 12	36 30 24 18	Numbers from 36 to 1, that are divisible at the same time by 2 and 3 without remainder are: 36, 30, 24, 18, 12 and 6. The number 12 is equal to the stop number, so we stop by 18.

Hints and Guidelines

The problem can be divided into **four** logical parts:

- **Reading** the input.
- **Checking** all numbers in the given range, and then running a **loop**.
- **Checking** the conditions of the problem according to every number in the given range.
- **Printing** the numbers.

First part is ordinary – we read **three** integer numbers from the console, so we will use **int**.

We have already seen examples of the **second** part – initialization of the **for** loop. It is a bit **tricky** – the explanation mentions that the numbers have to be printed in **reversed order**. This means that the **initial** value of the variable **i** will be **bigger**, and from the examples we can see that it is **M**. Thus, the **final** value of **i** should be **N**. The fact that we will print the results in reversed order and the values of **i**, suggests that the step would be **decreased by 1**.

for (int i = m; i >= n; i--)

After we have initialized the **for** loop, it is time for the **third** part of the problem – **checking** the condition if the given **number is divisible both by 2 and 3 without remainder**. We will do this using one simple **if** condition that we will leave to the reader to do by themselves.

Another **tricky** part of this problem is that apart from the above check we need to do **another** one – whether the **number is equal to the "stop" number** entered from the console on the third line. To do this check, the previous one has to be passed. For this reason, we will add another **if** statement that we will **nest in the previous one**. If the condition is **true**, we need to stop the program from printing. We can do this using a **break** operator, and it will lead us **out** of the **for** loop.

If the **condition** that checks whether the number is equal with "stop" number returns a **false** result, our program should **continue to print**. This covers the **fourth and last** part of our program.

Testing in the Judge System

Test your solution here: <https://judge.softuni.org/Contests/Practice/Index/515#2>.

Problem: Special Numbers

Write a program that reads one integer number N and generates all possible **special numbers** from **1111** to **9999**. To be considered **special**, a number must correspond to the **following condition**:

- N to be divisible by each of its digits without remainder.

Example: upon $N = 16$, **2418** is a special number:

- $16 / 2 = 8$ without remainder
- $16 / 4 = 4$ without remainder
- $16 / 1 = 16$ without remainder
- $16 / 8 = 2$ without remainder

Input Data

The input is read from the console and consists of **one integer** within the range **[1 ... 600 000]**.

Output Data

Print on the console **all special numbers**, separated by **space**.

Sample Input and Output

Input	Output	Comments
3	1111 1113 1131 1133 1311 1313 1331 1333 3111 3113 3131 3133 3311 3313 3331 3333	$3 / 1 = 3$ without remainder $3 / 3 = 1$ without remainder $3 / 3 = 1$ without remainder $3 / 3 = 1$ without remainder

Hints and Guidelines

Solve the problem by yourself using what you learned from the previous two problems. Keep in mind the difference between operators for **integer division** **/** and **division with remainder** **%** in C#.

Testing in the Judge System

Test your solution here: <https://judge.softuni.org/Contests/Practice/Index/515#3>.

Problem: Digits

Write a program that reads from the console an integer within the range **[100 ... 999]**, and then prints it a predefined number of times – modifying it before each print, as follows:

- If the number is divisible by **5** without remainder, **subtract** from it **its first digit**.
- If the number is divisible by **3** without remainder, **subtract** from it **its second digit**
- If none of the above-mentioned conditions is valid, **add** to it **its third digit**.

Print on the console **N lines**, and each line has **M numbers**, that are result of the above actions.

- $N = \text{sum of the first and second digit of the number}$.
- $M = \text{sum of the first and third digit of the number}$.

Input Data

The input is read from the **console** and is an integer within the range [100 ... 999].

Output Data

Print on the console **all integer numbers**, result of the above-mentioned calculations in the respective number of rows and columns as in the examples.

Sample Input and Output

Input	Output	Comments
376	382 388 394 400 397 403 409 415 412 418 424 430 427 433 439 445 442 448 454 460 457 463 469 475 472 478 484 490 487 493 499 505 502 508 514 520 517 523 529 535 532 538 544 550 547 553 559 565 562 568 574 580 577 583 589 595 592 598 604 610 607 613 619 625 622 628 634 640 637 643 649 655 652 658 664 670 667 673 679 685 682 688 694 700 697 703 709 715 712 718	10 lines with 9 numbers in each Input number 376 → neither 5, nor 3 → 376 + 6 → = 382 → neither 5, nor 3 → 382 + 6 = 388 + 6 = 394 + 6 = 400 → division by 5 → 400 - 3 = 397

Input	Output	Comments
132	129 126 123 120 119 121 123 120 119 121 123 120	$(1 + 3) = 4$ and $(1 + 2) = 3 \rightarrow$ 4 lines with 3 numbers in each Input number 132 → division by 3 → $132 - 3 =$ $= 129 \rightarrow$ division by 3 → $129 - 3 =$ $= 126 \rightarrow$ division by 3 → $126 - 3 =$ $= 123 \rightarrow$ division by 3 → $123 - 3 =$ $= 120 \rightarrow$ division by 5 → $120 - 1 =$ $121 \rightarrow$ neither by 5, nor 3 → $121 + 2 = 123$

Hints and Guidelines

Solve the problem **by yourself**, using what you learned from the previous ones. Remember that you will need to define **different** variables for each digit of the input number.

Testing in the Judge System

Test your solution here: <https://judge.softuni.org/Contests/Practice/Index/515#4>.

High-quality education profession and job for software engineers

- ✓ Free start for **absolute beginners**
- ✓ Choose among the **most in-demand** professions in the software industry
- ✓ Assistance for **career start**
- ✓ **Top trainers**, mentors and active learning **community**



SoftUni provides practical, modern and **innovative education** for programming, IT, design, marketing and digital skills for thousands of young people. The "**Software University**" program builds true **software engineering professionals**.

Join the end-to-end **software engineering learning program** at SoftUni to master the most in-demand programming concepts, languages, software technologies and skills through a modern teaching methodology, **interactive learning platform** and tons of practical exercises and projects. Our **curriculum** is developed together with the **IT companies** to closely match the industry demands.

SoftUni works directly with the **companies** from the software industry to **find a job** for its students and to make them successful software engineers.

The path of the student at SoftUni



Apply now

softuni.org/apply

Chapter 8.1. Practical Exam Preparation – Part I

In the present chapter we will examine a few **problems** with a level of **difficulty** that can be expected in the **problems** of the practical **exam** in "Programming Basics". We will **review** and **practice** all the knowledge that was gained from this book and through the "Programming Basics" course.

Video: Chapter Overview

Watch a video about what shall you learn in this chapter: <https://youtu.be/t0eqLOSzIR4>.

The "Programming Basics" Practical Exam

The course "Programming Basics" at SoftUni finishes with a **practical exam**. There are **6** problems included, and you will have **4 hours** to solve them. **Each** of the exam problems will **cover** one of the **topics** studied during the course. Problem topics are from the following 6 groups:

- Problem with **simple calculations** (no conditions)
- Problem with **simple condition** (simple checks)
- Problem with more **complex conditions** (nested checks and multiple checks)
- Problem with a **simple loop** (e.g. iterate from 1 to N)
- Problem with **nested loops** (e.g. drawing a 2D figure on the console)
- Problem with **nested loops** and **more complex logic** (loops and checks together)

Video: The Practical Exam Explained

Watch this video to learn more about the practical exam, the problems, the evaluation and the automated judge system: <https://youtu.be/KPFk980HWhw>.

The Online Evaluation System (Judge)

All **exams and exercises** from this book are automatically **tested** through the online Judge system: <https://judge.softuni.org>. For **each** of the problems there are **visible** (zero point) tests to help you understand what is expected of the problem and fix your mistakes, as well as **competition** tests that are **hidden** and check if your solution is working properly. In the **Judge** system you can log in with your **softuni.org** account.

How does the testing in the **Judge system** work? You **upload / paste** the source code of your solution from the judge Web page and you choose to compile and run it as a **C#** program. The program is then **tested** with a series of tests (input data + checks of the produced output), giving **points** for each **successful** test. For each problem the Judge provides a separate code submission page, separate evaluation tests and separate scoring (points).

Simple Calculations – Problems

The first problem of the "Programming Basics" Practical Exam covers **simple calculations without checks and loops**. Participants in the exam should know how to write simple programs, use **variables** and data, **read** and **print numbers** on the console and **perform simple calculations**. Let's solve a few **sample problems** with simple calculations.

Problem: Triangle Area

The first sample exam problem is about calculating the area of given triangle, specified by its coordinates (where the triangle stays horizontally).

Video: Triangle Area

Watch the video lesson about the triangle area problem: https://youtu.be/m2O8_rcNHtA.

Problem Description

Triangle in the plane is defined by the coordinates of its three vertices. First the vertex (x_1, y_1) is set. Then the other two vertices are set: (x_2, y_2) and (x_3, y_3) , which lie on a common horizontal line (i.e. they have the same Y coordinates). Write a program that calculates the area of the triangle by the coordinates of its three vertices.

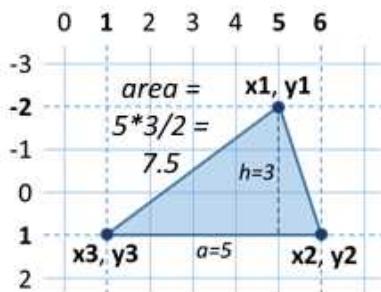
Input

The console gives 6 integers (one per line): $x_1, y_1, x_2, y_2, x_3, y_3$.

- All input numbers are in range $[-1000 \dots 1000]$.
- It is guaranteed that $y_2 = y_3$.

Output

Print on the console the area of the triangle.



Sample Input and Output

Input	Output	Visualization	Comments
5 -2 6 1 1 1	7.5		The side of the triangle: $a = 6 - 1 = 5$ The height of the triangle: $h = 1 - (-2) = 3$ Triangle area: $S = a * h / 2 = 5 * 3 / 2 = 7.5$

Input	Output	Visualization	Comments
4 1 -1 -3 3 -3	8		The side of the triangle: $a = 3 - (-1) = 4$ The height of the triangle: $h = 1 - (-3) = 4$ Triangle area: $S = a * h / 2 = 4 * 4 / 2 = 8$

Reading the Input Data

It is important in such types of tasks where some coordinates are given to pay attention to the order in which they are submitted, and to properly understand which of the coordinates we will use and how. In this case, the input is in order $x_1, y_1, x_2, y_2, x_3, y_3$. If we do not follow this sequence, the solution becomes wrong. First, we write the code that reads the input data:

```
int x1 = int.Parse(Console.ReadLine());
```

```
int y1 = int.Parse(Console.ReadLine());
int x2 = int.Parse(Console.ReadLine());
int y2 = int.Parse(Console.ReadLine());
int x3 = int.Parse(Console.ReadLine());
int y3 = int.Parse(Console.ReadLine());
```

Calculate Triangle Side and Height

We have to calculate **the side** and **the height** of the triangle. From the pictures, as well as the condition $y2 = y3$, we notice that the one **side** is always parallel to the horizontal axis. This means that its **length** is equal to the length of the segment between its coordinates **x2 and x3**, which is equal to the difference between the larger and the smaller coordinates. Similarly, we can calculate **the height**. It will always be equal to the difference between **y1 and y2** (or **y3**, as they are equal). Since we do not know if **x2** is greater than **x3**, or **y1** will be below or above the triangle side, we will use **the absolute values** of the difference to always get positive numbers, because one segment cannot have a negative length.

```
int a = Math.Abs(x2 - x3);
int h = Math.Abs(y2 - y1);
```

Calculate and Print Triangle Area

we will calculate it using our familiar formula for finding an **area of a triangle**. An important thing to keep in mind is that although we get only integers at an input, **the area** will not always be an integer. That's why we use a variable of **double** type for the area. We have to convert the right side of the equation, because if we give whole numbers as equation parameters, our result will also be an integer.

```
double s = (double)a * h / 2;
```

The only thing left is to print the area on the console.

```
Console.WriteLine(s);
```

Testing in the Judge System

Test your solution here: <https://judge.softuni.org/Contests/Practice/Index/516#0>.

Problem: Moving Bricks

The next sample exam problem is to calculate how many trolleys courses are needed to move given set of bricks (assuming the trolley has limited capacity).

Video: Moving Bricks

Watch the video lesson about solving the "Moving Bricks" problem: <https://youtu.be/NuPQOEPYjsI>.

Problem Description

Construction workers have to transfer a total of **x** **bricks**. **Workers** are **w** and work simultaneously. They transport the bricks in trolleys, each with a **capacity of m** bricks. Write a program that reads the integers **x**, **w**, and **m**, and calculates **what is the minimum number of courses** the workers need to do to transport the bricks.

Input

On the console **3 integers** are given, one per line:

- The **number of bricks x** is read from the first line.

- The number of workers' **w** is read from the second line
- The capacity of the trolley **m** is read from the third line.

All input numbers are integers in the range [1 ... 1000].

Output

Print on the console **the minimum number of courses** needed to transport the bricks.

Sample Input and Output

Input	Output	Comments
120 2 30	2	We have 2 workers, each transporting 30 bricks per course. In total, workers are transporting 60 bricks per course. To transport 120 bricks, exactly 2 courses are needed.

Input	Output	Comments
355 3 10	12	We have 3 workers, each transporting 10 bricks per course. In total, workers are transporting 30 bricks per course. To transport 355 bricks, exactly 12 courses are needed: 11 complete courses carry 330 bricks and the last 12th course carries the last 25 bricks.

Input	Output	Comments
5 12 30	1	We have 5 workers, each transporting 30 bricks per course. In total, workers are transporting 150 bricks per course. In order to transport 5 bricks, only 1 course is sufficient (although incomplete, with only 5 bricks).

Reading the Input Data

The input is standard, and we only need to be careful about the sequence in which we read the data.

```
int x = int.Parse(Console.ReadLine());
int w = int.Parse(Console.ReadLine());
int m = int.Parse(Console.ReadLine());
```

Calculating Bricks per Course

We calculate how many **bricks** the workers transport in a single course:

```
int bricksInOneCourse = w * m;
```

Calculating and Printing the Needed Courses

By dividing the total number of **bricks transported for 1 course**, we will obtain the number of **courses** required to carry them. We have to consider that when dividing whole numbers, the remainder is ignored and always rounded down. To avoid this, we will convert the right side of the equation to **double** and use the **Math.Ceiling(...)** function to round the result always up. When the bricks can be transferred with **an exact number of courses**, the division will return a whole number and there will be nothing to round. Accordingly, if not, the result of the division will be **the number of exact courses** but a decimal fraction. The decimal part will be rounded up and we will get the required **1 course** for the remaining bricks.

```
double totalCourses = Math.Ceiling((double)x / bricksInOneCourse);
```

Finally, we print the result on the console.

```
Console.WriteLine(totalCourses);
```

Testing in the Judge System

Test your solution here: <https://judge.softuni.org/Contests/Practice/Index/516#1>.

Simple Conditions – Problems

The second problem of the "Programming Basics" Practical Exam covers **conditional statements** and **simple calculations**. Participants in the exam should know how to implement simple logic using the **if-else conditional statements**, as well as how to read and write data from the console and perform simple calculations. Let's solve a few **sample problems** with simple conditions.

Problem: Point on a Segment

The next sample exam problem is about checking whether given point stays inside or outside given horizontal segment.

Video: Point on a Segment

Watch a video lesson about solving the "Point on a Segment" problem: <https://youtu.be/isrSHqLvV90>.

Problem Description

A **horizontal segment** is placed on a horizontal line, set with the **x** coordinates of both ends: **first** and **second**. A **point** is located **on** the same horizontal line and is set with its **x coordinate**. Write a program that checks whether the point is **inside or outside the segment** and calculates **the distance to the nearest end** of the segment.

Input

The console reads **3 integer numbers** (one per line):

- On the first line the number "first" is read – **one end of the segment**.
- On the second line the number "second" is read – **the other end of the segment**.
- On the third line the number "point" is read – **the location of the point**.

All input numbers are integers in the range [-1000 ... 1000].

Output

Print the result on the console:

- On the first line, print "in" or "out" – whether the point is inside or outside the segment.
- On the second line, print the distance from the point to the nearest end of the segment.

Sample Input and Output

Input	Output	Visualization
10 5 7	in 2	

Input	Output	Visualization
8 10 5	out 3	

Input	Output	Visualization
1 -2 3	out 2	

Reading the Input Data

We read the input from the console.

```
int first = int.Parse(Console.ReadLine());
int second = int.Parse(Console.ReadLine());
int point = int.Parse(Console.ReadLine());
```

Calculate the Minimum Distance to the Closest End

Since we do not know which **point** is on the left and which is on the right, we will create two variables to mark this. Since the **left point** is always the one with the smaller **x coordinate**, we will use **Math.Min(...)** to find it. Accordingly, the **right one** is always the one with a larger **x coordinate** and we will use **Math.Max(...)**. We will also find the distance from **point x** to the **two points**. Because we do not know their position relative to each other, we will use **Math.Abs(...)** to get a positive result.

```
int left = Math.Min(first, second);
int right = Math.Max(first, second);

int distanceLeft = Math.Abs(left - point);
int distanceRight = Math.Abs(right - point);
```

The shorter of the two **distances** we can found using **Math.Min(...)**.

```
int minDistance = Math.Min(distanceLeft, distanceRight);
```

Determining if Point is in or Out the Segment

What remains is to find whether the **point** is on or out of the line. The point will be **on the line** whenever it **matches** one of the other two points or its **x coordinate lies between them**. Otherwise, the point is **outside the line**. After checking, we display one of the two messages, depending on which condition is satisfied.

```
if (point >= left && point <= right)      else
{
    Console.WriteLine("in");                  {
}                                              Console.WriteLine("out");
}
```

Finally, we print the **distance** previously found.

```
Console.WriteLine(minDistance);
```

Testing in the Judge System

Test your solution here: <https://judge.softuni.org/Contests/Practice/Index/516#2>.

Problem: Point in a Figure

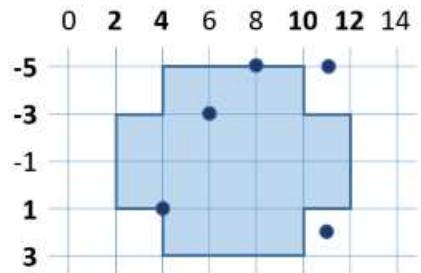
The next sample exam problem is about checking whether given point stays inside or outside given figure (see the image below).

Video: Point in a Figure

Watch a video lesson about solving the "Point in a Figure" problem: <https://youtu.be/SzxHyZVdhEo>.

Problem Description

Write a program that checks whether a point (with coordinates **x** and **y**) is **inside** or **outside** the figure, shown on the right.



Input

The console reads **two integers** (one per line): **x** and **y**.

All input numbers are integers in the range [-1000 ... 1000].

Output

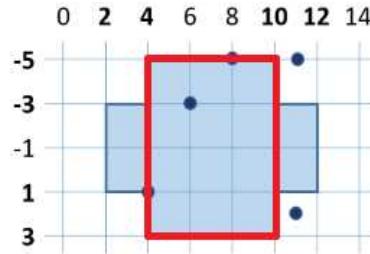
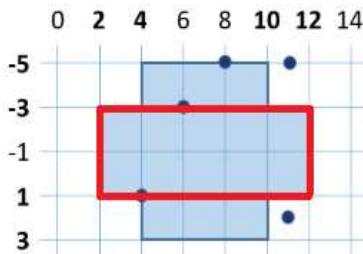
Print on the console "in" or "out" – whether the point is **inside** or **outside** the figure (the outline is inside).

Sample Input and Output

Input	Output	Input	Output	Input	Output	Input	Output
8 -5	in	6 -3	in	11 -5	out	11 2	out

Hints and Guidelines

To find out if the **point** is in the figure, we will divide **the figure** into 2 rectangles:



A sufficient condition is **the point** to be located in one of them, in order to be in **the figure**.

Determining the Point Location

We read the input data from the console:

```
int x = int.Parse(Console.ReadLine());
int y = int.Parse(Console.ReadLine());
```

We will initialize two variables that will mark whether **the point** is in one of the rectangles.

```
bool pointInRect1 = x >= 2 && x <= 12 && y >= -3 && y <= 1;  
bool pointInRect2 = x >= 4 && x <= 10 && y >= -5 && y <= 3;
```

When printing the message, we will check whether any of the variables has accepted a value of **true**. It is enough **only one** of them to be **true**, so that the point is in the figure.

```
if (pointInRect1 || pointInRect2)    else  
{                                {  
    Console.WriteLine("in");          Console.WriteLine("out");  
}
```

Testing in the Judge System

Test your solution here: <https://judge.softuni.org/Contests/Practice/Index/516#3>.

Complex Conditions – Problems

The **third** problem of the "Programming Basics" Practical Exam includes **several nested checks combined with simple calculations**. Participants in the exam should know how to work with **nested if-else statements** and how to use complex conditions, as well as how to read and print data from the console and perform calculations.

Let's solve a few **sample problems** with complex conditions.

Problem: Date After 5 Days

The next sample exam problem is about calculating the date 5 days after given date (day + month), having in mind that it might appear in the next month.

Video: Date After 5 Days

Watch a video lesson about solving the "Date After 5 Days" problem: https://youtu.be/01FEQP6r_xk.

Problem Description

There are two numbers **d** (day) and **m** (month) that form **a date**. Write a program that prints the date that will be **5 days a particular date**. For example, 5 days after **28.03** is the date **2.04**. We assume that the months: April, June, September and November have 30 days, February has 28 days, and the rest have 31 days. Months to be printed with **leading zero** when they contain a single digit (e.g. 01, 08).

Input

The input is read from the console and consists of two lines:

- On the first line we read an integer **d** in the range **[1 ... 31]** – day. The number of the day does not exceed the number of days in that month (e.g. 28 for February).
- On the second line we read an integer **m** in the range **[1 ... 12]** – month. Month 1 is January, month 2 is February, ..., month 12 is December. The month may contain a leading zero (e.g. April may be written as 4 or 04).

Output

Print a single line containing the date after 5 days in the format **day.month**. The month must be a 2-digit number with a leading zero (if necessary). The day must be printed without leading zero.

Sample Input and Output

Input	Output	Input	Output	Input	Output	Input	Output
28 03	2.04	27 12	1.01	25 1	30.01	26 02	3.03

Reading and Processing the Input Data

We take the input from the console.

```
int d = int.Parse(Console.ReadLine());
int m = int.Parse(Console.ReadLine());
```

To make our checks easier, we'll create a variable that will contain the **number of days** that we have in the month we set.

```
int daysInMonth = 31;      if (m == 4 || m == 6 || m == 9 || m == 11)
if (m == 2)                {
{                           daysInMonth = 30;
daysInMonth = 28;         }
}
```

Adding 5 Days

We are increasing **the day** by 5.

```
d += 5;
```

We check if **the day** has not exceeded the number of days in the **month**. If so, we must deduct the days of the month from the obtained day in order to calculate which day of the next month our day corresponds to.

```
if (d > daysInMonth)
{
    d -= daysInMonth;
}
```

After we have passed to the **next month**, this should be noted by increasing the initial one by 1. We need to check if it has not become greater than 12 and if it has, to adjust it. Because we cannot skip more than **one month** when we increase by 5 days, the following check is enough.

```
if (d > daysInMonth)
{
    d -= daysInMonth;
    m++;
    if (m > 12)
    {
        m = 1;
    }
}
```

Printing the Result

The only thing that remains is to print the result on the console.

It is important to **format the output** correctly to display the leading zero in the first 9 months. This is done by adding a **formatting string :D2** after the second element.

```
Console.WriteLine("{0}.{1:D2}", d, m);
```

Testing in the Judge System

Test your solution here: <https://judge.softuni.org/Contests/Practice/Index/516#4>.

Problem: Sums of 3 Numbers

The next sample exam problem is about checking multiple cases to find out if from given 3 numbers we can find two of them which sum up to the third of them.

Video: Sum of 3 Numbers

Watch the following video lesson to learn how to solve the "Sum of 3 Numbers" problem step by step: <https://youtu.be/7NwKXOtWXc>.

Problem Description

There are **3 integers** given. Write a program that checks if **the sum of two of the numbers is equal to the third one**. For example, if the numbers are **3, 5** and **2**, the sum of two of the numbers is equal to the third one: **$2 + 3 = 5$** .

Input

The console reads **three integers**, one per line. The numbers are in the range **[1 ... 1000]**.

Output

- Print a text line on the console containing the solution of the problem in the format "**a + b = c**", where **a, b** and **c** are among the three input numbers and **a ≤ b**.
- If the problem has no solution, print "**No**" on the console.

Sample Input and Output

Input	Output	Input	Output	Input	Output	Input	Output
3		2		1		2	
5	2 + 3 = 5	2	2 + 2 = 4	1	No	6	
2		4		5		3	No

Reading the Input Data

We take the input from the console.

```
int a = int.Parse(Console.ReadLine());
int b = int.Parse(Console.ReadLine());
int c = int.Parse(Console.ReadLine());
```

Composing a Template for the Solution

We can check if the **sum** of a pair of numbers is equal to the third number. We have 3 possible cases:

- $a + b = c$

- $a + c = b$
- $b + c = a$

We will write a **template**, which will later be complemented by the required code. If none of the above three conditions is met, we will make our program print "No".

```
if (a + b == c)
{
    // TODO
}
else if (a + c == b)
{
    // TODO
}
else if (b + c == a)
{
    // TODO
}
else
{
    Console.WriteLine("No");
}
```

Writing Code in the Template

We now have to understand the order in which the **two addends** will be written in the output of the program. For this purpose, we will create **a nested condition** that checks which one of the two numbers is the larger one. In the first case, it will look like this:

```
if (a + b == c)
{
    if (a > b)
    {
        Console.WriteLine("{0} + {1} = {2}", b, a, c);
    }
    else
    {
        Console.WriteLine("{0} + {1} = {2}", a, b, c);
    }
}
```

Similarly, we will supplement the other two cases. The full code of the program will look like this:

```
if (a + b == c)
{
    if (a > b)
    {
        Console.WriteLine("{0} + {1} = {2}", b, a, c);
    }
    else
    {
        Console.WriteLine("{0} + {1} = {2}", a, b, c);
    }
}
else if (a + c == b)
{
```

```
if (a < c)
{
    Console.WriteLine("{0} + {1} = {2}", a, c, b);
}
else
{
    Console.WriteLine("{0} + {1} = {2}", c, a, b);
}
}
else if (b + c == a)
{
    if (b < c)
    {
        Console.WriteLine("{0} + {1} = {2}", b, c, a);
    }
    else
    {
        Console.WriteLine("{0} + {1} = {2}", c, b, a);
    }
}
else
{
    Console.WriteLine("No");
}
```

Testing in the Judge System

Test your solution here: <https://judge.softuni.org/Contests/Practice/Index/516#5>.

Simple Loops – Problems

The fourth problem of the "Programming Basics" Practical Exam includes a **single loop with simple logic** in it. Participants in the exam should know how to work with simple **for-loops** and how to implement simple program logic with loops, how to read and process a sequence of numbers from the console and how to perform calculations and checks.

Let's solve a few **sample problems** with simple loops.

Problem: Sums with Step of 3

The next sample exam problem is about calculating 3 sums, holding the numbers from given sequence, staying at certain positions with step 3.

Video: Sums with Step of 3

Watch a video lesson to learn how to solve the "Sums with Step of 3" problem using a **for-loop** with several **if-else** statements inside: <https://youtu.be/bRHFuNNBmZc>.

Problem Description

We have given are n integers: a_1, a_2, \dots, a_n . Calculate the sums:

- $\text{sum1} = a_1 + a_4 + a_7 + \dots$ (the numbers are summed, starting from the first one with step of 3).
- $\text{sum2} = a_2 + a_5 + a_8 + \dots$ (the numbers are summed, starting from the second one with step 3).
- $\text{sum3} = a_3 + a_6 + a_9 + \dots$ (the numbers are summed, starting from the third one with step of 3).

Input

The input data is read from the console. The first line holds an integer n ($0 \leq n \leq 1000$). On the next n lines, we are given n integers in the range [-1000 ... 1000]: a_1, a_2, \dots, a_n .

Output

On the console we should print 3 lines containing the 3 sums in a format such as in the example.

Sample Input and Output

Input	Output
4	
7	sum1 = 19
-2	sum2 = -2
6	sum3 = 6
12	

Input	Output
5	
3	
5	sum1 = 10
2	sum2 = 13
7	
8	sum3 = 2

Input	Output
1	sum1 = 5
5	sum2 = 0
	sum3 = 0

Reading the Input Data

We will take the count of numbers from the console and declare starting values of the three sums.

```
int n = int.Parse(Console.ReadLine());  
  
int sum1 = 0;  
int sum2 = 0;  
int sum3 = 0;
```

Since we do not know in advance how many numbers we will process, we will take them one at a time in a loop which will be repeated n times and we will process them in the body of the loop.

```
for (int i = 0; i < n; i++)  
{  
    int a = int.Parse(Console.ReadLine());  
    //TODO  
}
```

Allocating Numbers and Printing Results

To find out in which of the three sums we need to add the number, we will divide its sequence number into three and we will use the remainder. We will use the variable i , which tracks the number of runs of the loop, in order to find out which sequence number we are at. When the remainder of $i/3$ is zero, it means we will add this number to the first sum, when it is 1 to the second one, and when it is 2 to the third one.

```
for (int i = 0; i < n; i++)
{
    int a = int.Parse(Console.ReadLine());
    if (i % 3 == 0)
    {
        sum1 += a;
    }
    if (i % 3 == 1)
    {
        sum2 += a;
    }
    if (i % 3 == 2)
    {
        sum3 += a;
    }
}
```

Finally, we will print the result on the console in the required format.

```
Console.WriteLine("sum1 = {0}", sum1);
Console.WriteLine("sum1 = {0}", sum2);
Console.WriteLine("sum1 = {0}", sum3);
```

Testing in the Judge System

Test your solution here: <https://judge.softuni.org/Contests/Practice/Index/516#6>.

Problem: Sequence of Increasing Elements

The next sample exam problem is about finding the longest increasing subsequence of given sequence of integers.

Video: Sequence of Increasing Elements

Watch a video lesson about the "Sequence of Increasing Elements" problem and its solution: <https://youtu.be/4ZHRC4usRAM>.

Problem Description

A series of n numbers is given: a_1, a_2, \dots, a_n . Calculate the length of the longest increasing sequence of consecutive elements in the series of numbers.

Input

The input data is read from the console. The first line holds an integer n ($0 \leq n \leq 1000$). On the following n lines, we are given n integers in the range $[-1000 \dots 1000]$: a_1, a_2, \dots, a_n .

Output

On the console we must print one number – the length of the longest increasing sequence.

Sample Input and Output

Input	Output	Input	Output	Input	Output	Input	Output
3		4		4		4	
5	2	2		1		5	
2		8	2	2	3	6	
4		7		4		7	
		6		4		8	2

Reading the Input Data and Creating Working Variables

To solve this problem, we need to think in a bit **more algorithmic way**. A sequence of numbers is given to us, and we need to check whether each **subsequent** one will be **larger than the previous one**, and if so, we count how long is the sequence in which this condition is fulfilled. Then we have to find **which sequence** of these is **the longest** one. To do this, let's create some variables that we will use during solving the problem.

```
int n = int.Parse(Console.ReadLine());
int countCurrentLongest = 0;
int countLongest = 0;
int aPrev = 0;
int a = 0;
```

The variable **n** is **the count of numbers** we get from the console. In **countCurrentLongest** we will keep **the number of elements** in the increasing sequence we are **currently counting**. For example, in the sequence: 5, 6, 1, 2, 3 **countCurrentLongest** will be 2 when we reach the **second element** of the counting (5, 6, 1, 2, 3) and will become 3 when we reach the **last element** (5, 6, 1, 2, 3), because the increasing row 1, 2, 3 has 3 elements. We will use **countLongest** to keep the **longest** increasing sequence. The other variables are **a** – the number we are **currently in**, and **aPrev** – the **previous number** which we will compare with **a** to see if the row is **growing**.

Determining Increasing Sequence

We begin to run the numbers and check if the present number **a** is larger than the previous **aPrev** one. If this is true, then the row is **growing**, and we need to increase its number by **1**. This is stored in the variable that tracks the length of the sequence we are currently in – **countCurrentLongest**. If the number **a** is **not greater** than the previous one, it means that a **new sequence** starts, and we have to start the count from **1**. Finally, after all the checks are done, **aPrev** becomes **the number** we are **currently** using, and we start the loop from the beginning with **the next** entered **a**.

Here is a sample implementation of the algorithm described:

```
for (int i = 0; i < n; i++)
{
    a = int.Parse(Console.ReadLine());

    if (a > aPrev)
    {
        countCurrentLongest++;
    }
    else
    {
        countCurrentLongest = 1;
```

```

    }
    aPrev = a;
}

```

Finding and Printing the Longest Sequence

What remains is to see which of all sequences is **the longest** one. We will do this by checking in the loop if **the sequence** we are **currently** in has become longer than the **longest one by now**. The whole loop will look like this:

```

for (int i = 0; i < n; i++)
{
    a = int.Parse(Console.ReadLine());

    if (a > aPrev)
    {
        countCurrentLongest++;
    }
    else
    {
        countCurrentLongest = 1;
    }

    if (countCurrentLongest > countLongest)
    {
        countLongest = countCurrentLongest;
    }
    aPrev = a;
}

```

Finally, we print the length of **the longest** sequence found.

```
Console.WriteLine(countLongest);
```

Testing in the Judge System

Test your solution here: <https://judge.softuni.org/Contests/Practice/Index/516#7>.

Drawing Figures – Problems

The **fifth** problem of the "Programming Basics" Practical Exam requires using one or several **nested loops** for **drawing a figure** on the console. Participants in the exam should know how to **think logically** and construct simple algorithms using calculations, checks and nested loops to solve problems, i.e. to think in an algorithmic way. This problem is more about algorithmic thinking than about coding and loops. Let's solve a few **sample problems** with nested loops (and drawing figures).

Problem: Perfect Diamond

The next sample exam problem is about using nested loops and calculations to print on the console a diamond of given size, like the ones shown below in the examples.

Video: Perfect Diamond

Watch the video lesson about the "Perfect Diamond" problem: https://youtu.be/UqQFpM_cgbY.

Problem Description

Write a program that reads an integer **n** from the console and draws a **perfect diamond** with size **n** as in the examples below.

Input

The input is an integer **n** within the range [1 ... 1000].

Output

The diamond should be printed on the console as in the examples below.

Sample Input and Output

Input	Output	Input	Output	Input	Output	Input	Output
2	<pre> * *- * </pre>	3	<pre> * *- *- * *- * </pre>	4	<pre> * *- *- *- * *- *- * </pre>	5	<pre> * *- *- *- *- * *- *- *- * </pre>

Hints and Guidelines

In tasks for drawing figures, the most important thing to consider is **the sequence** in which we will draw. Which items are **repeated** and with what **steps**? We can clearly see that **the top and bottom parts** of the diamond are the **same**. The easiest way to solve the problem is by creating **a loop** that draws the **upper part**, and then another **loop** that draws the **bottom part** (opposite to the top one).

Reading the Input Data

We will read the number **n** from the console.

```
int n = int.Parse(Console.ReadLine());
```

Printing the Top Part of the Diamond

We start painting the **top half** of the diamond. We clearly see that **each row** starts with a **few empty spaces** and *****. If we take a closer look, we will notice that **the empty spaces** are always equal to **n - the number of lines** (the first row is **n-1**, the second - **n-2**, etc.). We will start by drawing the number of **empty spaces**, and the **first asterisk**. Let's not forget to use **Console.WriteLine(...)** instead of **Console.WriteLine(...)** to stay on **the same line**. At the end of the line we write **Console.WriteLine(...)** to go to **a new line**. Notice that we start counting from **1**, not from **0**. Next, we will only add a few times **-*** to **finish the line**.

Here is part of the code for the **top of the diamond**:

```
for (int i = 1; i <= n; i++)
{
    Console.Write(new string(' ', n - i));
    Console.Write("*");
}
```

```
// TODO: Draw the rest of the line
Console.WriteLine();
}
```

What remains is to **complete each line** with the required number of `-*` elements. On each row we have to add `i-1` such **items** (on the first $1-1 \rightarrow 0$, the second $\rightarrow 1$, etc.)

Here is the complete code for drawing **the top of the diamond**:

```
for (int i = 1; i <= n; i++)
{
    Console.Write(new string(' ', n - i));
    Console.Write("*");
    for (int j = 0; j < i - 1; j++)
    {
        Console.WriteLine("-*");
    }
    Console.WriteLine();
}
```

Printing the Bottom Part of the Diamond

To draw the **bottom part** of the diamond, we have to reverse **the upper part**. We will count from `n-1`, because if we start from `n`, we will draw the middle row twice. Do not forget to change **the step** from `++` to `--`.

Here is the code for drawing **the bottom part of the diamond**:

```
for (int i = n - 1; i >= 1; i--)
{
    Console.Write(new string(' ', n - i));
    Console.Write("*");
    for (int j = 1; j < i; j++)
    {
        Console.WriteLine("-*");
    }
    Console.WriteLine();
}
```

What remains is **to assemble the whole program** by first reading the input, printing the top part of the diamond and then the bottom part of the diamond.

Testing in the Judge System

Test your solution here: <https://judge.softuni.org/Contests/Practice/Index/516#8>.

Problem: Rectangle with Stars in the Center

The next sample exam problem is about using nested loops and calculations to print on the console a rectangle of given size with stars in the middle, like the ones shown below in the examples.

Video: Rectangle with Stars in the Center

Watch the video lesson about solving the "Rectangle with Stars in the Center" problem step by step:
<https://youtu.be/6cOJDJm6sNk>.

Problem Description

Write a program that reads from the console an integer **n** and draws a **rectangle** with size **n** with two asterisks **is its center** as in the examples below.

Input

The input is an integer **n** in the range [2 ... 1000].

Output

The rectangle should be printed on the console as in the examples below.

Sample Input and Output

Input	Output	Input	Output	Input	Output	Input	Output
2	%%%%% %**% %%%%%	3	%%%%% % % % ** % % % %%%%%	4	%%%%% % % % ** % % % %%%%%	5	%%%%%%%%%% % % % % % ** % % % %%%%%%%%%%

Reading the Input Data

We read the input data.

```
int n = int.Parse(Console.ReadLine());
```

Printing the First and the Last Rows

The first thing we can easily notice is that **the first and last rows** contain $2 * n$ symbols **%**. We will start with this and then draw the middle part of the rectangle.

```
Console.WriteLine(new string('%', n * 2));  
// TODO: Draw middle of the rectangle  
Console.WriteLine(new string('%', n * 2));
```

Printing the Middle Rows

From the examples we see that **the middle** part of the figure always has **odd number** of rows. Note that when an **even number** is set, the number of rows is equal to **the previous odd** number ($2 \rightarrow 1, 4 \rightarrow 3$, etc.). We create a variable that represents the number of rows that our rectangle will have and correct it if the number **n is even**. Then we will draw a **rectangle without the asterisks**. Each row has for **the beginning and the end** the symbol **%** and between them $2 * n - 2$ empty spaces (the width is $2 * n$ and we subtract 2 for the two percent at the end). Do not forget to move the code for the **last line after the loop**.

```
int numRows = n;  
if (n % 2 == 0)  
{  
    numRows--;  
}
```

```
}
```



```
for (int i = 0; i < numRows; i++)
{
    Console.WriteLine("%");
    Console.Write(new string(' ', n * 2 - 2));
    // TODO: Place the stars
    Console.WriteLine("%");
    Console.WriteLine();
}
```

We can start and test the code so far. Everything without the two asterisks in the middle should work correctly.

Adding Stars in the Center of the Rectangle

Now, in the body of the loop let's add the asterisks. We'll check if we're on the middle row. If we are in the middle, we will draw the row together with the asterisks, if not – we will draw a normal row. The line with the asterisks has **n-2 empty spaces** (n is half the length and we remove the asterisk and the percent) and again **n-2 empty spaces**. We leave out of the check the two percent at the beginning and at the end of the row.

```
for (int i = 0; i < numRows; i++)
{
    Console.WriteLine("%");
    if (i == numRows / 2)
    {
        Console.Write(new string(' ', n - 2));
        Console.Write("##");
        Console.Write(new string(' ', n - 2));
    }
    else
    {
        Console.Write(new string(' ', n * 2 - 2));
    }
    Console.WriteLine("%");
    Console.WriteLine();
}
```

Testing in the Judge System

Test your solution here: <https://judge.softuni.org/Contests/Practice/Index/516#9>.

Nested Loops – Problems

The last (sixth) problem of the "Programming Basics" Practical Exam requires using of **several nested loops and more complex logic inside them**. The problems examine participants' ability to **think in an algorithmic way** and to solve non-trivial coding problems that require nested loops with more complex logic and calculations, along with reading and printing data on the console.

Let's solve a few **sample problems** with nested loops and more complex logic.

Problem: Increasing 4 Numbers

The next sample exam problem is about using nested loops and program logic to generate all possible combinations of 4 increasing numbers in given range.

Video: Increasing 4 Numbers

Watch the following video lesson to learn how to solve the "Increasing 4 Numbers" problem:
<https://youtu.be/2DuNHqmbP5Y>.

Problem Description

For given pair of numbers **a** and **b** generate all four number **n₁**, **n₂**, **n₃**, **n₄**, for which **a ≤ n₁ < n₂ < n₃ < n₄ ≤ b**.

In combinatorics such a selection of subset from given set (or range) is called "[combination](#)" (<https://en.wikipedia.org/wiki/Combination>), so the problem is essence is to **generate all combinations** of 4 elements from given range of integers.

Sample Input and Output

Input	Output	Input	Output	Input	Output
3 7	3 4 5 6 3 4 5 7 3 4 6 7 3 5 6 7 4 5 6 7	5 7	No	10 13	10 11 12 13

Input

The input contains two integers **a** and **b** in the range [0 ... 1000], one per line.

Output

The output contains all **numbers in batches of four**, in ascending order, one per line.

Reading the Input Data

We will read the input data from the console. We also create the additional variable **count**, which will keep track of **existing number ranges**.

```
int a = int.Parse(Console.ReadLine());
int b = int.Parse(Console.ReadLine());
int count = 0;
```

Implementation with 2 Numbers

We will most easily solve the problem if we logically divide it **in parts**. If we are required to draw all the rows from a number between **a** and **b**, we will do it using **one loop** that takes all the numbers from **a** to **b**. Let's think how to do this with **series of two numbers**. The answer is easy – we will use **nested loops** like this:

```

for (int i = a; i <= b; i++)
{
    for (int j = i + 1; j <= b; j++)
    {
        Console.WriteLine("{0} {1}", i, j);
    }
}

```

We can test the incomplete program to see if it's accurate so far. It must print all pairs of numbers **i**, **j** for which **i** ≤ **j**.

Since each **next number** of the row must be **greater than the previous one**, the second loop will run around **i + 1** (the next greater number). Accordingly, if **there is no sequence** of two incremental numbers (**a** and **b** are equal), the second loop **will not be fulfilled**, and nothing will be printed on the console.

Implementation with 4 Numbers

Similarly, what remains is to implement **the nested loops** for four numbers. We will add an **increase of the counter** that we initialized in order to know if **there is such a sequence**.

```

for (int i = a; i <= b; i++)
{
    for (int j = i + 1; j <= b; j++)
    {
        for (int k = j + 1; k <= b; k++)
        {
            for (int l = k + 1; l <= b; l++)
            {
                Console.WriteLine("{0} {1} {2} {3}", i, j, k, l);
                count++;
            }
        }
    }
}

```

Finally, we check if **the counter** is equal to **0** and we will print "No" on the console accordingly, if so.

```

if (count == 0)
{
    Console.WriteLine("No");
}

```

Testing in the Judge System

Test your solution here: <https://judge.softuni.org/Contests/Practice/Index/516#10>.

Problem: Generating Rectangles

The next sample exam problem is about using nested loops and program logic to generate all possible rectangles, which have integer coordinates in given range and given minimum area.

Video: Generating Rectangles

Watch a video about solving the "Generating Rectangles" problem: <https://youtu.be/gk1OpMXpCc>.

Problem Description

By a given number **n** and a minimum area **m**, generate all possible rectangles with integer coordinates in the range $[-n \dots n]$ with an area of at least **m**. The generated rectangles must be printed in the following format:

- (**left, top**) (**right, bottom**) -> **area**

Rectangles are defined using the top left and bottom right corner. The following inequalities are in effect:

- $-n \leq \text{left} < \text{right} \leq n$
- $-n \leq \text{top} < \text{bottom} \leq n$

Sample Input and Output

Input	Output	Input	Output	Input	Output
1 2	(-1, -1) (0, 1) -> 2 (-1, -1) (1, 0) -> 2 (-1, -1) (1, 1) -> 4 (-1, 0) (1, 1) -> 2 (0, -1) (1, 1) -> 2	2 17	No	3 36	(-3, -3) (3, 3) -> 36

Input

Two numbers, one per line, are entered from the console:

- An integer **n** in the range [1 ... 100] – sets the minimum and maximum coordinates of a peak.
- An integer **m** in the range [0 ... 50 000] – sets the minimum area of the generated rectangles.

Output

- The described rectangles should be printed on the console in a format such as in the examples below.
- If there are no rectangles for the specified **n** and **m**, then print "No".
- The order of rectangles in the output is not important, so use and order of your choice.

Reading the Input Data

Read the input data from the console. We will also create a **counter**, which will store the number of rectangles found.

```
int n = int.Parse(Console.ReadLine());
int m = int.Parse(Console.ReadLine());
int count = 0;
```

Sample Idea for the Solution

It is very important to be able to imagine the problem before we begin to solve it. In our case it is required to search for rectangles in a coordinate system. The thing we know is that the **left point** will always have the coordinate **x**, smaller than **the right** one. Accordingly, **the upper one** will always have

a smaller **y** coordinate than **the lower one**. To find all the rectangles, we'll have to create **a loop** similar to the previous problem, but this time, **not every next loop** will start from **the next number** because some of **the coordinates** can be equal (for example **left** and **top**).

```
for (int left = -n; left < n; left++)
{
    for (int top = -n; top < n; top++)
    {
        for (int right = left + 1; right <= n; right++)
        {
            for (int bottom = top + 1; bottom <= n; bottom++)
            {
                // TODO
            }
        }
    }
}
```

With the variables **left** and **right** we will follow the coordinates **horizontally** and with **top** and **bottom** – **vertically**.

Calculating the Rectangle Area and Printing the Output

The important thing here is knowing the corresponding coordinates so we can correctly calculate the sides of the rectangle. Now we have to find **the area of the rectangle** and check if it is **greater than or equal** to **m**. One **side** will be **the difference between left and right** and **the other one** – **between top and bottom**. Since the coordinates may be eventually interchanged, we will use **absolute values**. Again, we add **the counter** in the loop, counting **only the rectangles** we write. It is important to note that the writing order is **left, top, right, bottom**, as it is set in the problem's description.

```
for (int left = -n; left < n; left++)
{
    for (int top = -n; top < n; top++)
    {
        for (int right = left + 1; right <= n; right++)
        {
            for (int bottom = top + 1; bottom <= n; bottom++)
            {
                int area = Math.Abs(right - left)
                          * Math.Abs(bottom - top);
                if (area >= m)
                {
                    Console.WriteLine("{0}, {1}) ({2}, {3}) -> {4}",
                                      left, top, right, bottom, area);
                    count++;
                }
            }
        }
    }
}
```

Finally, we print "No" if there are no such rectangles.

```
if (count == 0)
{
    Console.WriteLine("No");
}
```

Testing in the Judge System

Test your solution here: <https://judge.softuni.org/Contests/Practice/Index/516#11>.

Practical Exam Preparation – Summary

The "Programming Basics" **final exam** consists of **6 problems** for **4 hours**. Each of the exam problems will **cover** one of the **topics** studied in the previous chapters. The 6 problems are from the following **6 categories**:

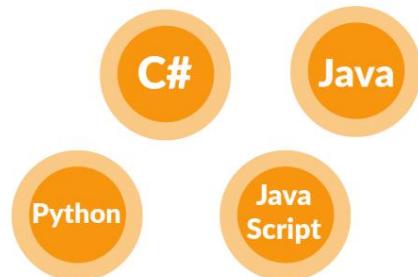
- Problem with **simple calculations** (no conditions)
- Problem with **simple condition** (simple checks)
- Problem with more **complex conditions** (nested checks and multiple checks)
- Problem with a **simple loop** (e.g. iterate from 1 to N)
- Problem with **nested loops** (e.g. drawing a 2D figure on the console)
- Problem with **nested loops** and **more complex logic** (loops and checks together)

Video: Chapter Summary

Watch a video about what we have learned in this chapter: https://youtu.be/p4J22J04K_E.

High-quality education profession and job for software engineers

- ✓ Free start for **absolute beginners**
- ✓ Choose among the **most in-demand** professions in the software industry
- ✓ Assistance for **career start**
- ✓ **Top trainers**, mentors and active learning **community**



SoftUni provides practical, modern and **innovative education** for programming, IT, design, marketing and digital skills for thousands of young people. The "**Software University**" program builds true **software engineering professionals**.

Join the end-to-end **software engineering learning program** at SoftUni to master the most in-demand programming concepts, languages, software technologies and skills through a modern teaching methodology, **interactive learning platform** and tons of practical exercises and projects. Our **curriculum** is developed together with the **IT companies** to closely match the industry demands.

SoftUni works directly with the **companies** from the software industry to **find a job** for its students and to make them successful software engineers.

The path of the student at SoftUni



Apply now

softuni.org/apply

Chapter 8.2. Practical Exam Preparation – Part II

In the current chapter we will review a practical exam in “Programming Basics” conducted at SoftUni on December 18, 2016. The problems will give you a good overview of what you can expect at an admission exam in Programming at SoftUni. The exam covers the material studied in the current book and the Programming Basics course at SoftUni (<https://softuni.org/curriculum>).

Types of Exam Problems

Traditionally, the admission exam at SoftUni consists of **6 practical problems in programming** of the following types:

- Simple problems (no conditions).
- A problem with a single condition.
- A problem with more complex conditions.
- A problem with a single loop.
- A problem with nested loops (drawing a figure on the console).
- A problem with nested loops and more complex logic.

Let's examine a **real exam topic**: the 6 exam problems and their solutions.

Problem: Distance

Write a program that calculates **what is the distance passed by a car (in kilometers)**, if we know the **initial speed** (km/h), the **initial time frame** in minutes, then the **speed is increased by 10%**, the **second time frame**, then the **speed is decreased by 5%**, and the **time until the end of the trip**. In order to calculate the distance, you need to **convert the minutes into hours** (e.g. 70 minutes = 1.1666 hours).

Input Data

The input comes from the console and consists of **4 lines**:

- The **initial speed** in km/h – an integer within the range [1 ... 300].
- The **first time frame** in minutes – an integer within the range [1 ... 1000].
- The **second time frame** in minutes – an integer within the range [1 ... 1000].
- The **third time frame** in minutes – an integer within the range [1 ... 1000].

Output Data

Print a number on the console: **the kilometers passed**, formatted up to the **second digit after the decimal point**.

Sample Input and Output

Input	Output	Comments
90 60 70 80	330.90	Distance with initial speed: $90 \text{ km/h} * 1 \text{ hour (60 min)} = 90 \text{ km}$ After speed increase: $90 + 10\% = 99.00 \text{ km/h} * 1.166 \text{ hours (70 min)} = 115.50 \text{ km}$ After speed decrease: $99 - 5\% = 94.05 \text{ km/h} * 1.33 \text{ hours (80 min)} = 125.40 \text{ km}$ Total number of km passed: 330.9 km

Input	Output	Comments
140 112 75 190	917.12	Distance with initial speed: $140 \text{ km/h} * 1.86 \text{ hours (112 min)} = 261.33 \text{ km}$ After speed increase: $140 + 10\% = 154.00 \text{ km/h} * 1.25 \text{ hours (75 min)} = 192.5 \text{ km}$ After speed decrease: $154.00 - 5\% = 146.29 \text{ km/h} * 3.16 \text{ hours (190 min)} = 463.28 \text{ km}$ Total number of km passed: 917.1166 km

Hints and Guidelines

It is possible that such a description may look **misleading** and incomplete at first glance, which adds to the **complexity** of a relatively easy task. Let's **separate** the problem into a few **sub-problems** and try to **solve** each of them one by one, which will lead us to the final result:

- Our **initial** sub-problem will be to **read the input data** entered by the user and **store them in appropriate variables**.
- **Execution** of the main programming **logic**, which in our case is a batch of simple calculations of the properties that we already have.
- **Calculation** and shaping up the end **result**.

The **main** part of the programming logic is to **calculate** what will be the **distance** passed after all **changes** in speed. As during **execution** of the program, part of the **data** that we have **is modified**, we could **separate** the program **code** into a few **logically separated parts**:

- **Calculation** of the **distance** passed with initial speed.
- Change of **speed** and calculation of the **distance** passed.
- Last change of **speed** and **calculation**.
- Summing up.

Reading the Input Data

We use the following function to **read** the data from the **console**:

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

By definition, the **input data** is given as **four** separate lines. This is why we need to execute **the previous code** **four** times in total.

```
string initialSpeedString = Console.ReadLine();
// string firstIntervalString = TODO
// string secondIntervalString = TODO
// string thirdIntervalString = TODO
```

Selecting Data Type for Calculations

In order to **perform** the **calculations**, we select **decimal** type.



The data type for real numbers with decimal representation in **C#** is the 128-bit **decimal** type. It has the **accuracy** of **28 to 29** decimal numbers. Its **minimum** value is **-7.9×10²⁸**, and its **maximum** value is **+7.9×10²⁸**. Its default value is **0.0m** or **0.0M**. The **m** symbol at the end explicitly indicates that the number is **decimal** type

(by **default** all real numbers are **double** type). The numbers closest to **0** that can be stored in **decimal** are $\pm 1.0 \times 10^{-28}$. It is evident that **decimal** cannot store **very large** positive and negative numbers (for example with hundreds of digits), nor values **very close to 0**. On the other hand, this type rarely causes **any errors** upon financial calculations because it represents the numbers as a **sum of the power of the number 10**, upon which the **round-off errors** are much **less** compared to when we use binary representation. Real numbers of **decimal** type are exceptionally **suitable for monetary calculations** – calculation of incomes, liabilities, taxes, interest, etc.

This way we solved successfully the **first sub-problem**.

Converting the Input Data

The next step is to **convert the input data** into appropriate **types**, in order to be able to perform the needed calculations. We select **Int32** or **int** as an appropriate type, because the condition of the problem says that the input data must be within a **particular range**, for which this data type is completely sufficient. We will do the **conversion** in the following way:

```
int initialSpeed = int.Parse(initialSpeedString);
// int firstInterval = TODO
// int secondInterval = TODO
// int thirdInterval = TODO
```

Helper Variable

We initially **store** one **variable** that will be used multiple times. This centralization approach gives us **flexibility** and **possibility** to **modify** the end result of the program with minimum efforts. In case we need to change the value, we must do it in **only once place in the code**, which saves us time and effort.

```
decimal minutesPerHour = 60m;
```



Avoiding repetitive code (centralization of the program logic) in the tasks that we examine in the present book may look unnecessary at first glance, but this approach is very important upon building large applications in a real work environment, and its exercising in an initial stage of training will help you build a quality programming style.

Calculating Travel Distance

We calculate the **travel time** (in hours) by **dividing the time by 60** (minutes in an hour). The **travel distance** is calculated by **multiplying the starting speed by the time passed** (in hours). After that we change the speed by increasing it by **10%**, as per the task description. Calculating the **percentage**, as well as the following **distances** passed, is done in the following way:

- The **time frame** (in hours) is calculated by **dividing** the provided time frame in minutes by the minutes that are contained in an hour (60).
- The **distance passed** is calculated by **multiplying** the time frame (in hours) by the speed that is obtained after the increase.
- The next step is to **decrease the speed** by **5%**, as per the problem description.
- We calculate the **remaining distance** in the manner described in the first two points.

This is **sample code**, which implements the above described steps:

```
decimal firstIntervalHours = firstInterval / minutesPerHour;
decimal firstDistance = initialSpeed * firstIntervalHours;

decimal speedAfterIncrease = initialSpeed
    + ((initialSpeed * 10) / 100m);
decimal secondIntervalHours = secondInterval / minutesPerHour;
decimal secondDistance = speedAfterIncrease * secondIntervalHours;
```

Calculating and Printing the Output

Up until now we were able to **solve two** of the **most important sub-problems**, namely the **data input** and their processing. What remains is to **calculate the end result**. As by the description we are required to **format it** up to 2 symbols after the decimal point, we can do this in the following **manner**:

```
decimal finalDistance =
    firstDistance + secondDistance + thirdDistance;
decimal finalResult = string.Format("{0:f2}", finalResult);
Console.WriteLine(finalResult);
```

If you worked accurately and wrote the program using the input data given in the task description, you will be convinced that it works properly.

Testing in the Judge System

Test your solution here: <https://judge.softuni.org/Contests/Practice/Index/517#0>.

Problem: Changing Tiles

Harry has some **savings** that he wants to use to **change the tiles** on the bathroom floor. The floor is **rectangular**, and the **tiles** are **triangular**. Write a **program** that **calculates** if his **savings** will be sufficient. Read from the **console** the **width** and **length** of the **floor**, as well as **one of the sides** of the **triangle** with its **height towards it**. We must **calculate** how many **tiles** are **needed**, in order to cover the **floor**. The **number** of **tiles** **must be rounded up** to the **higher integer** and **5 more tiles** **must be added** as **spare tiles**. Also **read from the console** – the **price per tile** and the **amount paid for the work of a workman**. All prices and money calculations are performed in **1v** (Bulgarian levs, BGN).

Input Data

The following 7 lines must be read from the console:

- Savings.
- Floor width.
- Floor length.
- Side of the rectangle.
- Height of the rectangle.
- Price of a tile in **1v** (Bulgarian levs, BGN).
- Fee to be paid to the workman.

All numbers must be real numbers within the range [0.00 ... 5000.00].

Output Data

The following must be printed on the console as a **single line**:

- If the money is sufficient: “[Remaining funds] lv left.”
- If the money IS NOT sufficient: “You’ll need {Insufficient funds} lv more.”

The result must be formatted up to the second digit after the decimal point.

Sample Input and Output

Input	Output	Comments
500 3 2.5 0.5 0.7 7.80 100	25.60 lv left.	Floor area → $3 * 2.5 = 7.5$ Tile area → $0.5 * 0.7 / 2 = 0.175$ Needed tiles → $7.5 / 0.175 = 42.857\dots = 43 + 5$ spare tiles = 48 Total amount → $48 * 7.8 + 100$ (workman) = 474.4 $474.4 < 500 \rightarrow 25.60$ lv left

Input	Output	Comments
1000 5.55 8.95 0.90 0.85 13.99 321	You'll need 1209.65 lv more.	Floor area → $5.55 * 8.95 = 49.67249$ Tile area → $0.9 * 0.85 / 2 = 0.3825$ Needed tiles → $49.67249 / 0.3825 = 129.86\dots = 130 + 5$ spare tiles = 135 Total amount → $135 * 13.99 + 321$ (workman) = 2209.65 $2209.65 > 1000 \rightarrow 1209.65$ lv are insufficient

Hints and Guidelines

The following task requires our problem to accept more input data and to perform a larger number of calculations, despite the fact that the solution is **identical**. Accepting the input data is done in the **familiar way**. Pay attention that the **Input** part of the condition states that all input data must be in **real numbers**, and for that reason we would use **decimal** type.

Now that we already have everything for executing the programming logic, we can move to the following part. How can we **calculate** what is the **needed** number of tiles that will be sufficient to cover the entire floor? The requirement is that tiles have **triangular** shape, which can cause confusion, but practically, the task needs just **simple calculations**. We can calculate the **common part of the floor** by the formula for finding rectangle area, as well as the **area of a single tile** using the relevant formula for triangle area.

In order to calculate the **number of tiles** that are needed, **we divide the floor area by the area of a single tile** (we should not forget to add the 5 additional tiles, that were mentioned in the requirements).



Pay attention that the requirements state that we should round up the number of tiles, obtained upon the division, up to the higher number, and then we should add 5. Find more information about the system function that does that: **Math.Ceiling(...)**.

We can find the end result by **calculating the total amount** that is needed to cover the entire floor, by **summing up the tile price and the price that will be paid to the workman**, that we have from the input data. We can figure out that the **total costs** for tiles can be calculated by **multiplying the number of tiles by the price per tile**. We will find out whether the amount that we have will be sufficient by comparing the savings (based on the input data) and the total costs.

Testing in the Judge System

Test your solution here: <https://judge.softuni.org/Contests/Practice/Index/517#1>.

Problem: Flowers Shop

A flowers shop offers 3 types of flowers: **chrysanthemums**, **roses** and **tulips**. The prices depend on the season.

Season	Chrysanthemums	Roses	Tulips
spring / summer	2.00 USD/pc	4.10 USD/pc	2.50 USD/pc
autumn / winter	3.75 USD/pc	4.50 USD/pc	4.15 USD/pc

On holidays, prices of all flowers are **increased by 15%**. The following **discounts** are offered:

- For purchasing more than 7 tulips in spring – **5% of the price** of the whole bouquet.
- For purchasing 10 or more roses in winter – **10% of the price** of the whole bouquet.
- For purchasing more than 20 flowers in total in any season – **20% of the price** of the whole bouquet.

Discounts are made in the above described order and can be combined! All discounts are valid after increasing of the price on a holiday!

The price for arranging a bouquet is always **2 USD**. Write a program that calculates the **price of a bouquet**.

Input Data

The input is read from the **console** and contains **exactly 5 lines**:

- The first line holds **the number of purchased chrysanthemums** – an integer in range [0 ... 200].
- The second line holds **the number of purchased roses** – an integer within the range [0 ... 200].
- The third line holds **the number of purchased tulips** – an integer within the range [0 ... 200].
- The fourth line indicates **the season** – [Spring, Summer, Autumn, Winter].
- The fifth line specifies **if the day is a holiday** – [Y = yes / N = no].

Output Data

Print on the console 1 number – **the price of flowers**, formatted up to the second digit after the decimal point.

Sample Input and Output

Input	Output	Comments
2 4 8 Spring Y	46.14	Price: $2 \cdot 2.00 + 4 \cdot 4.10 + 8 \cdot 2.50 = 40.40$ USD Holiday: $40.40 + 15\% = 46.46$ USD 5% discount for more than 7 tulips in spring: 44.14 The flowers are in total 20 or less: no discount $44.14 + 2$ for arranging the bouquet = 46.14 USD

Input	Output	Comments
3 10 9 Winter N	69.39	Price: $3 \cdot 3.75 + 10 \cdot 4.50 + 9 \cdot 4.15 = 93.60$ USD Not a holiday: no increase in price 10% discount for 10 or more roses in winter: 84.24 The flowers are in total over 20: 20% discount = 67.392 $67.392 + 2$ for arranging the bouquet = 69.392 USD

Hints and Guidelines

We will divide the problem into smaller sub-problems, as described below.

Separating the Constant Values in Variables

After carefully reading the requirements, we understand that once again we need to do **simple calculations**, however this time we will need **additional logical conditions**. We need to pay more attention to the moment of **making changes** in the final price, in order to be able to properly build the logic of our program. Again, the bold text gives us sufficient **guidelines** on how to proceed. For a start, we will **separate** the already **defined** values in **variables**, like we did in the previous tasks:

```
// Initial price list
decimal roseAutumnWinterPrice = 4.50m;
decimal roseSpringSummerPrice = 4.10m;
decimal tulipAutumnWinterPrice = 4.15m;
decimal tulipSpringSummerPrice = 2.50m;
decimal chrysantemumAutumnWinterPrice = 3.75m;
decimal chrysantemumSpringSummerPrice = 2m;
decimal arrangePrice = 2m;
```

We will also do the same for the rest of the defined values:

```
// Price increases
int priceIncreasePercentage = 15;

// Price decreases
int tulipPriceDecreasePercentage = 5;
int rosePriceDecreasePercentage = 10;
int totalPriceDecreasePercentage = 20;

// Price decrease thresholds
int tulipPriceDecreaseThreshold = 7;
int rosePriceDecreaseThreshold = 10;
int totalPriceDecreaseThreshold = 20;
```

Reading the Input Data

Our next sub-task is to **read** properly **the input** data from the console. We will do that in the familiar way, but this time we will **combine two** separate functions – one for **reading** a line from the console and another one for its **conversion** into a numeric data type:

```
int chrysantemumsPurchased = int.Parse(Console.ReadLine());
// int rosesPurchased = TODO
// int tulipsPurchased = TODO
// string season = TODO
// string isSpecialDay = TODO
```

Preparing the Program Logic

Let's think of the most appropriate way to **structure** our programming logic. By the requirements it becomes clear that the path of the program is divided mainly into two parts: **spring / summer** and **autumn / winter**. We can do the separation by conditional statement, by storing variables in advance for the **prices** of the individual flowers, as well as for the **end result**.

```
if (season == "Winter" || season == "Autumn")
{
    rosesPrice = rosesPurchases * roseAutumnWinterPrice;
    chrysantemumsPrice = chrysantemumsPurchases
        * chrysantemumAutumnWinterPrice;
    tulipsPrice = tulipsPurchased * tulipAutumnWinterPrice;
    totalCost = rosesPrice + chrysantemumsPrice + tulipsPrice;
}
else
{
    // TODO
}
```

What remains is to perform **a few checks** regarding the **discounts** of the different types of flowers, depending on the season, and to modify the end result.

Testing in the Judge System

Test your solution here: <https://judge.softuni.org/Contests/Practice/Index/517#2>.

Problem: Grades

Write a program that **calculates statistics for grades** in an exam. At the beginning, the program reads the **number of students** who attended the exam and for **each student – their grade**. At the end, the program must **print the percentage of students** that have grades between 2.00 and 2.99, between 3.00 and 3.99, between 4.00 and 4.99, 5.00 or more, as well as the **average grade** of the exam.

Note: we use the **Bulgarian grading system**, where the grade scale starts from **2.00** (Fail) and ends at **6.00** (Excellent): https://en.wikipedia.org/wiki/Grading_systems_by_country#Bulgaria.

Input Data

Read from the console a **sequence of numbers**, each on a separate line:

- On the first line – the **number of students** who attended the exam – an integer within the range [1 ... 1000].
- For **each individual student** on a separate line – the **grade on the exam** – a real number within the range [2.00 ... 6.00].

Output Data

Print on the console **5 lines** that hold the following information:

- "Top students: {percentage of students with grade of 5.00 or more}%".
- "Between 4.00 and 4.99: {between 4.00 and 4.99 included}%".
- "Between 3.00 and 3.99: {between 3.00 and 3.99 included}%".
- "Fail: {less than 3.00}%".
- "Average: {average grade}".

The results must be formatted up to the second digit after the decimal point.

Sample Input and Output

Input	Output	Comments
6 2 3 4 5 6 2.2	Top students: 33.33% Between 4.00 and 4.99: 16.67% Between 3.00 and 3.99: 16.67% Fail: 33.33% Average: 3.70	5 or more: 2 students = 33.33% of 6 Between 4.00 and 4.99: 1 student = 30% of 6 Between 3.00 and 3.99: 1 student = 20% of 6 Below 3: 2 students = 20% of 6 The average grade is: $2 + 3 + 4 + 5 + 6 + 2.2 = 22.2 / 6 = 3.70$

Input	Output	Comments
10 3.00 2.99 5.68 3.01 4 4 6.00 4.50 2.44 5	Top students: 30.00% Between 4.00 and 4.99: 30.00% Between 3.00 and 3.99: 20.00% Fail: 20.00% Average: 4.06	5 or more: 3 students = 30% of 10 Between 4.00 and 4.99: 3 students = 30% of 10 Between 3.00 and 3.99: 2 students = 20% of 10 Below 3: 2 students = 20% of 10 The average grade is: $3 + 2.99 + 5.68 + 3.01 + 4 + 4 + 6 + 4.50 + 2.44 + 5 = 40.62 / 10 = 4.062$

Hints and Guidelines

We will divide the problem into smaller sub-problems, as described below.

Reading the Input Data and Creating Helper Variables

By requirements we see that **first** we will read the **number** of students, and then, **their grades**. For that reason, **firstly** in an **int** variable we will read the **number** of students. In order to read and process the grades themselves, we will use a **for** loop. The value of the **int** variable will be the **end** value of the **i** variable from the loop. This way, **all** iterations of the loop will read **each one of the grades**.

```
int numberOfStudents = int.Parse(Console.ReadLine());
for (int i = 0; i < numberOfStudents; i++)
{
    // TODO
}
```

Before executing the code of the **for** loop, we will create variables where we will store **the number of students** for each group: poor results (up to 2.99), results from 3 to 3.99, from 4 to 4.99 and grades above 5. We will also need one more variable, where we will store **the sum of all grades**, via which we will calculate the average grade of all students.

```
double numberOfFailedStudents = 0;
double numberOfAverageStudents = 0;
double numberOfGoodStudents = 0;
double numberOfExcellentStudents = 0;
double totalResult = 0;
```

Allocating Students into Groups

We run the **loop** and inside it we **declare one more** variable, in which we will store the **currently entered grade**. The variable will be **double** type and upon each iteration we will check **what is its value**. According to this value, **we increase** the number of students in the relevant group by **1**, as we should not forget to also increase the **total** amount of the grades, which we also track.

```
for (int i = 0; i < numberOfStudents; i++)
{
    double grade = double.Parse(Console.ReadLine());
    totalResult += grade;
    if (grade < 3)
    {
        numberOfFailedStudents++;
    }
    else // TODO: check other groups
}
```

We can calculate what **percentage** is taken by a particular **group of students** from the total number by multiplying the number of students in the relevant group by **100** and then dividing this by the **total** number of students.



Pay attention to the numeric data type that you work with upon doing these calculations.

The **end result** is formed in the well known fashion – **up to the second digit** after the decimal point.

Testing in the Judge System

Test your solution here: <https://judge.softuni.org/Contests/Practice/Index/517#3>.

Problem: Christmas Hat

Write a program that reads from the console an **integer n** and draws a **Christmas hat** with width of **4 * n + 1 columns** and height of **2 * n + 5 rows**, as in the examples below.

Input Data

The input is read from the console – an **integer n** within the range [3 ... 100].

Output Data

Print on the console a **Christmas hat**, exactly like in the examples.

Sample Input and Output

Input	Output	Input	Output
4	<pre>...../ \.\ /.....***.....*-*-*.....*--*--*.....*---*---*.....*---*---*... ..*---*---*.. .*---*---*.. *---*---*.. *****... *.*.*.*.*.*.*. * *****...</pre>	7	<pre>...../ \.....\ /.....***.....*-*-*.....*--*--*.....*---*---*.....*---*---*...</pre>

Problem Analysis

In tasks requiring **drawing** on the console, most often the user inputs **an integer** that is related to the **total size of the figure** that we need to draw. As the task requirements mention how the total length and width of the figure are calculated, we can use them as **starting points**. In the examples it is clear that regardless of the input data, we always have **first two rows** that are almost identical.

```
...../|\. .....
.....\|/.....
```

We also notice that the **last three rows** are always present, as **two** of them are completely **the same**.

```
*****
*.*.*.*.*.*.*. *
*****
```

By these observations we can come up with the **formula** for the **height of the variable part** of the Christmas hat. We use the formula specified in the task to calculate the total height, by subtracting the size of the unchangeable part. We obtain $(2 * n + 5) - 5$ or $2 * n$.

Drawing the Dynamic Part of the Figure

To draw the **dynamic** or the variable part of the figure, we will use a **loop**. The size of the loop will be from **0** to the **width** that we have by requirements, namely $4 * n + 1$. Since we will use this formula in **a few places** in the code, it is a good practice to declare it in a **separate variable**. Before running the loop, we should **declare variables** for the **number** of individual symbols that participate in the dynamic part: **dots** and **dashes**. By analyzing examples, we can also prepare formulas for the **starting values** of these variables. Initially, the **dashes** are **0**, but it is clear that we can calculate the number of **dots** by

subtracting **3** from the **total width** (the number of symbols that are building the top of the Christmas hat) and then **dividing by 2**, as the number of dots on both sides of the hat is the same.

```
.....***.....  
.....*-*.....  
....*---*....  
....*---*....  
...*-----*...  
..*-----*..  
.-----*..  
*-----*
```

What remains is to execute the body of the loop, as **after each** drawing we **decrease** the number of dots by **1** and **increase the number of dashes** by **1**. Let's not forget to draw one **star** between each of them. The sequence of drawing in the body of the loop is the following:

- Symbol string of dots
- Star
- Symbol string of dashes
- Star
- Symbol string of dashes
- Star
- Symbol string of dots

In case we have worked properly, we will obtain figures identical to those in the examples.

Testing in the Judge System

Test your solution here: <https://judge.softuni.org/Contests/Practice/Index/517#4>.

Problem: Letters Combination

Write a program that prints on the console **all combinations of 3 letters** within a specified range, by skipping the combinations **containing certain letter**. Finally, print the number of printed combinations.

Sample Input and Output

Input	Output	Comments
a c b	aaa aac aca acc caa cac cca ccc 8	All possible combinations with letters 'a', 'b' and 'c' are: aaa aab aac aba abb abc aca acb acc baa bab bac bba bbb bbc bca bcb bcc caa cab cac cba cbb cbc cca ccb ccc The combinations containing 'b' are not valid. 8 valid combinations remain.

Input	Output
a c z	aaa aab aac aba abb abc aca acb acc baa bab bac bba bbb bbc bca bcb caa cab cac cba cbb cbc cca ccb ccc 27

Input	Output
f k h	fff ffg ffi ffj ffk fgf fgg fgi fgi fgk fif fig fii fij fik fff fjj fjk fkf fkg fki fkj fkk gff gfg gfi gfj gfk ggf ggg ggi ggj ggk gif gig gii gij gik gif ggg gii gjk gkf gkg gki gjk gkk iff ifg ifi ifj ifk ifg ifg iji igj igk iif iig iii iij iik iif iig iji ijk ikf ikg iki ijk ijk jff jfg jfi jff jfk jgf jgg jgi jgj jgk jif jig jii jij jik jjf jjg jji jjj jjk jkf jkg jki jkj jkk kff kfg kfi kfj kfk kgf kgg kgi kgj kgk kif kig kii kij kik kjf kjg kji kjj kjk kkf kkg kki kkj kkk 125

Input Data

The input is read from the **console** and contains **exactly 3 lines**:

- A small letter from the English alphabet for a **beginning of the range** – between 'a' and 'z'.
- A small English letter for the **end of the range** – between the **first letter** and 'z'.
- A small English letter – from 'a' to 'z' – as the combinations containing this letter are **skipped**.

Output Data

Print on a single line **all combinations** corresponding to the requirements, followed by **their number**, separated by a space.

Reading the Input Data

By requirements, we have input data on **3 lines**, each of which is represented by one character of the **ASCII table** (<http://www.asciitable.com>). We could use an already **defined function** in C#, by converting the input data into **char** data type, as follows:

```
char startLetter = char.Parse(Console.ReadLine());
// TODO
```

Printing All Characters from Start to End

Let's think of how we can achieve the **end result**. In case the task requirement is to print all characters, from the starting to the end one (by skipping a particular letter), what should we do?

The easiest and most efficient way is to use a **loop**, by passing through **all characters** and printing those that are **different** from the **letter** that we need to skip. One of the advantages of C# is that we have the opportunity to use a different data type for a loop variable:

```
for (char i = 'a'; i <= 'z'; i++)
{
    Console.Write(i + " ");
}
```

The result of running the code is all letters from **a** to **z** included, printed on a single line and separated by spaces. Does this look like the end result of our task? We must find a **way** to print **3 characters**, as required, instead of **1**. Running such a program very much looks like a slot machine. We often win in slots, if we arrange a few identical characters on a row. Let's say that the machine has space for three characters. When we **stop** on a particular **character** on the first place, the other two places will **continue** rolling characters among all possible ones. In our case, **all possible characters** are the letters from the starting to the end one, entered by the user, and the solution of our program is identical to the way a slot machine works.

Printing Combination of 3 Characters

We use a **loop** that runs through **all characters** from the starting to the end letter (included). On **each iteration** of the **first** loop, we run a **second** one with the same parameters (but **only if** the letter of the first loop is valid, i.e. does not match the one that we must exclude, by requirements). In each iteration of the **second** loop, we run **one** more with the **same parameters** and the same **condition**. This way we have three nested loops, as we will print the characters in the body of the **latter**.

```
for (char i = startLetter; i <= endLetter; i++)
{
    if (i != exceptLetter)
    {
        // TODO
    }
}
```

Let's not forget that we also need to print the **total number of valid combinations** that we have found, and they must be printed on the **same line**, separated by a space.

Testing in the Judge System

Test your solution here: <https://judge.softuni.org/Contests/Practice/Index/517#5>.

Chapter 9.1. Problems for Champions – Part I

In this chapter, we will offer the reader a few **more difficult tasks** that aim to develop **algorithmic skills** and acquire **programming techniques** to solve tasks with higher complexity.

More Complex Problems on the Studied Material

We will solve together several programming problems that cover the material studied in the book, but more difficult than the usual problems of the entrance exams at SoftUni. If you want to become a **champion on the basics of programming**, we recommend training to solve such complex tasks to make it easy for you to take exams.

Problem: Crossing Sequences

We have two sequences:

- a sequence of **Tribonacci** (by analogy with the Fibonacci sequence), where each number is the **sum of the previous three** (with given three numbers)
- a sequence generated by a **numerical spiral** defined by looping like a spiral (right, bottom, left, top, right, bottom, left, top, etc.) of a matrix of numbers starting from its center with a given starting number and incremental step, by storing the current numbers in the sequence each time we make a turn

Write a program that finds the first number that appears in both sequences defined in the aforementioned way.

Example

Let the **Tribonacci sequence** start with **1, 2 and 3**. This means that the **first sequence** will contain the numbers **1, 2, 3, 6, 11, 20, 37, 68, 125, 230, 423, 778, 1431, 2632, 4841, 8904, 16377, 30122, 55403, 101902**, and so on.

At the same time, let the **numbers in the spiral** begin with **5** and the spiral increases by **2** at each step.

Then the **second sequence** will contain the numbers **5, 7, 9, 13, 17, 23, 29, 37** and so on. We see that **37** is the first number to be found in the Tribonacci sequence and in the spiral one, and that is the desired solution to the problem.

45	55
...	17	19	21	23
...	15	5	7
...	13	11	9
37	29
...	65

Input Data

Input data should be read from the console.

- On the first three lines of the input we will read **three integers**, representing the **first three numbers** in the Tribonacci sequence, positive non-zero numbers, sorted in ascending order.
- On the next two lines of the input we will read **two integers** representing the **first number** and the **step** for each cell of the matrix for the spiral of numbers. The numbers representing the spiral are positive non-zero numbers.

Input data will always be valid and will always be in the format described. No need to check.

Output Data

The result should be printed on the console.

On the single line of the output, we must print **the smallest number that occurs in both sequences**. If there is no number in the range [1 ... 1 000 000], which can be found in both sequences, print "No".

Constraints

- All numbers in the input will be in the range [1 ... 1 000 000].
- Allowed program time: 0.3 seconds.
- Allowed memory: 16 MB.

Sample Input and Output

Input	Output	Input	Output	Input	Output	Input	Output
1		13		99		4	
2		25		99		1	
3	37	99	13	99	No	7	
5		5		2		23	
2		2		2		3	71

Hints and Guidelines

The problem seems quite complicated, so we will break it into simpler sub-problems.

Processing the Input

The first step in solving the problem is to read and process the input. Input data consists of **5 integers**: 3 for the Tribonacci sequence and 2 for the numerical spiral.

```
int tribonacciFirst = int.Parse(Console.ReadLine());
// TODO: Read remaining numbers
```

Once we have the input data, we need to think about how we will generate the numbers in the two sequences.

Generating Tribonacci Sequence

For the Tribonacci sequence we will always **collect the previous three values** and then move the values of those numbers (the three previous ones) one position in the sequence, i.e. the value of the first one must accept the value of the second one, and so on. When we are done with the number, we will store its value in **an array**. Since the problem description states that the numbers in the sequences do not exceed 1,000,000, we can stop generating this range at exactly 1,000,000.

```
var tribonacciNumbers = new List<int>()
{
    tribonacciFirst,
    tribonacciSecond,
    tribonacciThird };

var tribonacciCurrent = tribonacciThird;

while (tribonacciCurrent < 1000000)
{
    tribonacciCurrent = tribonacciFirst
        + tribonacciSecond + tribonacciThird;
```

```

    tribonacciNumbers.Add(tribonacciCurrent);

    tribonacciFirst = tribonacciSecond;
    tribonacciSecond = tribonacciThird;
    tribonacciThird = tribonacciCurrent;
}

```

Generating Numerical Spiral

We need to think of **a relation** between numbers in the numerical spiral so we can easily generate every next number without having to look at matrices and loop through them. If we carefully look at the picture from the description, we will notice that **every 2 "turns" in the spiral, the numbers we skip are increased by 1**, i.e. from 5 to 7 and from 7 to 9, not a single number is skipped, but we directly add with **the step** of the sequence. From 9 to 13 and from 13 to 17 we skip a number, i.e. we add the step twice. From 17 to 23 and from 23 to 29 we skip two numbers, i.e. we add the step three times and so on.

Thus, we see that for the first two we have **the last number + 1 * the step**, the next two we add with the **2 * the step** and so on. Every time we want to get to the next number of the spiral, we will have to make such calculations.

```
spiralCurrent += spiralStep * spiralStepMul;
```

What we have to take care of is **for each two numbers, our multiplier** (let's call it "coefficient") **must increase by 1 (spiralStepMul++)**, which can be achieved with a simple check (**spiralCount % 2 == 0**). The whole code from the generation of the spiral in **an array** is given below.

```

var spiralNumbers = new List<int>() { spiralCurrent };
var spiralCount = 0;
var spiralStepMul = 1;

while (spiralCurrent < 1000000)
{
    spiralCurrent += spiralStepMul * spiralStepMul;

    spiralNumbers.Add(spiralCurrent);
    spiralCount++;
    if (spiralCount % 2 == 0)
    {
        spiralStepMul++;
    }
}

```

Finding Common Number for the Sequences

Once we have generated the numbers in both sequences, we can proceed to unite them and build the final solution. How will it look? For **each of the numbers** in the first sequence (starting from the smaller one) we will check if it exists in the other one. The first number that meets this criterion will be **the answer** to the problem.

We will do a **linear** search in the second array, and we will leave the more curious participants to optimize it using the technique called **binary search** because the second array is generated in sorted form, i.e. it meets the requirement to apply this type of search. The code for finding our solution will look like this:

```

var found = false;

for (int i = 0; i < tribonacciNumbers.Count; i++)
{
    for (int j = 0; j < spiralNumbers.Count; j++)
    {
        if (tribonacciNumbers[i] == spiralNumbers[j] &&
            tribonacciNumbers[i] <= 1000000)
        {
            Console.WriteLine(tribonacciNumbers[i]);
            found = true;
            break;
        }
    }

    if (found)
    {
        break;
    }
}

if (!found)
{
    Console.WriteLine("No");
}

```

Alternative Solution

The previous solution to the problem uses arrays to store the values. Arrays are not needed to solve the problem. There is an **alternative solution** that generates the numbers and works directly with them instead of keeping them in an array. On **every step** we can check whether **the numbers in the two sequences match**. If this is the case, we will print the number on the console and terminate the execution of our program. Otherwise, we will see the current number of **which sequence is the smaller one** and we will generate the next one where we are "lagging". The idea is that **we will generate numbers from the sequence that is "behind"** until we skip the current number of the other sequence and then vice versa, and if we find a match in the meantime, we will terminate the execution.

```

while (tribonacciCurrent <= 1000000 && spiralCurrent <= 1000000)
{
    if (tribonacciCurrent == spiralCurrent)
    {
        // TODO: Print and stop execution
    }
    else if (tribonacciCurrent < spiralCurrent)
    {
        // TODO: Generate next Tribonacci number
    }
    else
    {
        // TODO: Generate next Spiral number
    }
}

```

Testing in the Judge System

Test your solution here: <https://judge.softuni.org/Contests/Practice/Index/518#0>.

Problem: Magic Dates

Date is given in a "dd-mm-yyyy" format, e.g. 17-04-2018. We calculate **the weight of that date** by taking all of its digits, multiplying each digit with the others after it, and finally summing up all the results obtained. In our case, we have 8 digits: 17032007, so the weight is $1*7 + 1*0 + 1*3 + 1*2 + 1*0 + 1*0 + 1*7 + 7*0 + 7*3 + 7*2 + 7*0 + 7*0 + 7*7 + 0*3 + 0*2 + 0*0 + 0*0 + 0*7 + 3*2 + 3*0 + 3*7 + 2*0 + 2*0 + 2*7 + 0*0 + 0*7 + 0*7 = 144$.

Our task is to write a program that finds all the **magical dates between two specific years (inclusively) corresponding to given weight**. Dates must be printed in ascending order (by date) in the format "dd-mm-yyyy". We will only use the valid dates in the traditional calendar (the leap years have 29 days in February).

Sample Input and Output

Input	Output
2007	17-03-2007
2007	13-07-2007
144	31-07-2007

Input	Output
2003	
2004	
1500	No

Input	Output
2011	01-01-2011
2012	10-01-2011
14	01-10-2011
	10-10-2011

Input	Output
2012	09-01-2013
2014	17-01-2013
80	23-03-2013
	11-07-2013
	01-09-2013
	10-09-2013
	09-10-2013
	17-10-2013
	07-11-2013
	24-11-2013
	14-12-2013
	23-11-2014
	13-12-2014
	31-12-2014

Input Data

Input data should be read from the console. It consists of 3 lines:

- The first line contains an integer: **start year**.
- The second line contains an integer: **end year**.
- The third line contains an integer: **the search weight** for the dates.

Input data will always be valid and will always be in the format described. No need to check.

Output Data

The result should be printed on the console as consecutive dates in "**dd-mm-yyyy**" format, sorted by date in ascending order. Each string must be in a separate line. If there are no existing magic dates, print "No".

Constraints

- The start and final year are integer numbers in the range [1900 ... 2100].
- Magic weight is an integer in the range [1 ... 1000].
- Allowed program time: 0.25 seconds.
- Allowed memory: 16 MB.

Hints and Guidelines

We start with the input data. In this case, we have **3 integers** that should be read from the console, as this is the only entry and processing of input for the problem.

Having the start and the end year, it is nice to know how we will go through every date, not to worry about how many days there are in the month and whether it is a leap year, and so on.

Loop through Dates

For looping through the dates, we will take advantage of the functionality that gives us the **DateTime** class in **C#**. We will define a **start date variable** that we can do using the constructor that accepts a year, month, and day. We know the year is the starting year we read from the console and the month and the day must be January and 1st respectively. In C#, the "constructor" of **DateTime** accepts as first argument the year, as second argument the month and as third argument the day of the month:

```
DateTime currentDate = new DateTime(startYear, 1, 1);
```

Once we have the start date, we want to create a **loop that runs until we exceed the final year** (or until we pass December 31 in the final year if we compare the full dates), increasing each day by 1 day.

To increase by one day in each rotation, we will use a method of **DateTime - .AddDays(...)**, which will add one day to the current date. The method will take care instead of us, to decide where to skip the next month, how many days there is a month and everything around the leap years.

```
currentDate = currentDate.AddDays(1);
```

Caution: since the **.AddDays(...)** method returns the "new" date, it is important to assign the result, not just to call the method!

In the end, our loop may look like this:

```
while (currentDate.Year <= endYear)
{
    // TODO: Do all the magic
    currentDate = currentDate.AddDays(1);
}
```

Note: we can achieve the same result with a **for loop**: the **initialization** of the date goes to the first part of the **for**, the **condition** is preserved, and the **step** is the increase by 1 day.

Calculating Date Weight

Each date consists of exactly **8 characters (digits)** – **2 for the day (d1, d2)**, **2 for the month (d3, d4)** and **4 for the year (d5 to d8)**. This means that we will always have the same calculation every time, and we can benefit from this **to define the formula statically** (i.e. not to use loops, referring to different numbers from the date, but write the whole formula). To be able to write it, we will need **all digits from the date** in individual variables to make all the necessary multiplications. By using the division

and partition operations on the individual components of the date, using the **Day**, **Month** and **Year** properties, we can retrieve each digit.

```
int d1 = currentDate.Day / 10; // First day digit
int d2 = currentDate.Day % 10; // Second day digit

int d3 = currentDate.Month / 100; // First month digit
int d4 = currentDate.Month % 100; // Second month digit

int d5 = currentDate.Year / 1000; // First year digit
int d6 = (currentDate.Year / 100) % 10; // Second year digit
int d7 = currentDate.Year % 1000; // Third year digit
int d8 = currentDate.Year % 10; // Fourth year digit
```

Let's also explain one of the more interesting lines here. Let's take the second digit of the year for example (**d6**). We divide the year by 100, and we take a remainder of 10. What do we do? First, we eliminate the last 2 digits of the year by dividing by 100 (Example: **2018/100 = 20**). With the remainder of 10, we take the last digit of the resulting number (**20 % 10 = 0**) and so we get 0, which is the second digit of 2018.

What remains is to do the calculation that will give us the magical weight of a given date. In order **not to write all multiplications** as shown in the example, we will simply apply a grouping. What we need to do is multiply each digit with those that follow it. Instead of typing **d1 * d2 + d1 * d3 + ... + d1 * d8**, we can shorten this expression to **d1 * (d2 + d3 + ... + d8)** for grouping when we have multiplication and summing up. Applying the same simplification for the other multiplications, we get the following formula:

```
dateWeight = d1 * (d2 + d3 + d4 + d5 + d6 + d7 + d8) +
// d2 * ... +
// ...
```

Printing the Output

Once we have the weight calculated of a given date, we need **to check and see if it matches the magical weight we want**, in order to know if it should be printed or not. Checking can be done using a standard **if** block, taking care to print the date in the correct format.

```
if (dataWeight == numberToSearchFor)
{
    // Print
    found = true;
}
```

To print the dates, we have two options:

- The first way is to use the **.ToString(...)** method, where we can **submit the date format**, i.e. whether the days will be printed with a leading zero or not, whether the months will be printed with leading zeros or not, in words or digits, with a short or full name, etc.

```
Console.WriteLine(currentDate.ToString("Date Format Goes Here));
```

- The second option is to take the individual components of the date **Day**, **Month** and **Year** as we did when calculating, and to form the output by **formatting string**.

```
Console.WriteLine($"{0}-{1}-{2}", /*Date components goes here*/);
```

Caution: as we go through the dates from the start year to the end one, they will always be arranged in ascending order as per the description.

Finally, if we have not found a date that is eligible, we will have a **false** value in the **found** variable and we will be able to print **No**.

```
if (!found)
{
    Console.WriteLine("No");
}
```

Testing in the Judge System

Test your solution here: <https://judge.softuni.org/Contests/Practice/Index/518#1>.

Problem: Five Special Letters

Two numbers are given: **start** and **end**. Write a program that generates all combinations of 5 letters, each among the sets of `{'a', 'b', 'c', 'd', 'e'}` so that the weight of these 5 letters is a number in the range `[start ... end]`, inclusive. Print them in alphabetical order, in a single row, separated by a space. The weight of the letters is calculated as follows:

```
weight('a') = 5;
weight('b') = -12;
weight('c') = 47;
weight('d') = 7;
weight('e') = -32;
```

The weight of the sequence of the letters `c1, c2, ..., cn` is calculated by removing all the letters that are repeated (from right to left) and then calculating the formula:

```
weight(c1c2...cn) = 1 * weight(c1) + 2 * weight(c2) + ... + n * weight(cn)
```

For example, the weight of **bcddc** is calculated as follows:

First, we remove the repeating letters and get **bcd**. Then we apply the formula: $1 * \text{weight}('b') + 2 * \text{weight}('c') + 3 * \text{weight}('d') = 1 * (-12) + 2 * 47 + 3 * 7 = 103$.

Another example: `weight("cadae") = weight("cade") = 1 * 47 + 2 * 5 + 3 * 7 + 4 * (-32) = -50`.

Input Data

The input data is read from the console. It consists of two numbers:

- The number for **start**.
- The number for **end**.

Input data will always be valid and will always be in the format described. No need to check.

Output Data

The result should be printed on the console as a sequence of strings, arranged in alphabetical order. Each string must be separated from the next one by a single space. If the weight of any of the 5 letter strings does not exist within the specified range, print "No".

Constraints

- Numbers for **start** and **end** are integers in the range [-10000 ... 10000].
- Allowed program time: 0.25 seconds.
- Allowed memory: 16 MB.

Sample Input and Output

Input	Output	Comments
40 42	bcead bdcea	weight("bcead") = 41 weight("bdcea") = 40

Input	Output
300	
400	No

Input	Output
-1 1	bcdea cebda eaaad eaada eaadd eaade eaaaed eadaa eadad eadae eadda eaddd eadde eadea eaded eadee eaeed eaeda eaedd eaeede eaeed eeaad eead a eeadd eead eaeed eeead

Input	Output
200 300	baadc babdc badac badbc badca badcb badcc badcd baddc bbadc bbdac bdaac bdabc bdaca bdacb bdacc bdacd bdadc bdbac bddac beadc bedac eabdc ebadc ebdac edbac

Hints and Guidelines

Let's give some hints and guidelines for solving this problem.

Reading the Input Data

As every problem, we start the solution by **reading and processing the input data**. In this case, we have **two integers** that can be processed with a combination of the **int.Parse(...)** and **Console.ReadLine()** methods.

```
int firstNumber = int.Parse(Console.ReadLine());
int secondNumber = int.Parse(Console.ReadLine());
```

We have several main points in the problem – **generating all combinations** with a length of 5 including the 5 letters, **removing repeating letters** and **calculating weight** for a simplified word. The answer will consist of every word whose weight is within the given range [**firstNumber**, **secondNumber**].

Generating All Combinations

In order to generate **all combinations with length of 1** using 5 symbols, we would use a **loop from 0 to 4**, as we want each number of the loop to match one character. In order to generate **any combinations of length 2** using 5 characters (i.e. "aa", "ab", "ac", ..., "ba", ...), we would create **two nested loops each running through the digits from 0 to 4**, as we will once again make sure that each digit matches a specific character. We will repeat this step 5 times, so we will finally have 5 nested loops with indexes **i1, i2, i3, i4** and **i5**.

```
for (int i1 = 0; i1 < 5; i1++)
{
    for (int i2 = 0; i2 < 5; i2++)
    {
        for (int i3 = 0; i3 < 5; i3++)
        {
            for (int i4 = 0; i4 < 5; i4++)
            {
                for (int i5 = 0; i5 < 5; i5++)
                {
                    // Process the combination i1, i2, i3, i4, i5
                }
            }
        }
    }
}
```

```

    {
        for (int i4 = 0; i4 < 5; i4++)
        {
            for (int i5 = 0; i5 < 5; i5++)
            {
            }
        }
    }
}

```

Transforming Combinations into Words

Now that we have all 5-digit combinations, we must find a way to "turn" the five digits into a word with the letters from 'a' to 'e'. One of the ways to do that is to **predefine a simple string that contains the letters** that we have

```
string pattern = "abcde";
```

and for each digit we take the letter from the particular position. This way, the number 00000 will become "aaaaa", and the number 02423 will become "acecd". We can create the 5-letter string in the following way.

```
string fullWord = "" + pattern[i1]
                + pattern[i2]
                + pattern[i3]
                + pattern[i4]
                + pattern[i5];
```

Another way: we can convert the digits to letters by using their arrangement in the ASCII table. The expression '**a**' + **i** will return 'a' in case **i** = 0, 'b' in case **i** = 1, 'c' in case **i** = 2, etc.

This way we already have generated all 5-letter combinations and can proceed with the following part of the task.

Attention: as we have chosen a '**pattern**' that takes into consideration the alphabetical arrangement of the letters, and cycles are run in the appropriate manner, the algorithm will generate the words in alphabetical order and there is no need for additional sorting before printing the output.

Removing Repetitive Letters

Once we have the finished string, we have to remove all the repeating symbols. We will do this by adding **the letters from left to right in a new string and each time before adding a letter, we will check if it already exists** – if it does, we will skip it and if it doesn't, we will add it. To begin with, we will add the first letter to the starting string.

```
string word = pattern[i1].ToString();
```

Then we will do the same with the other 4, checking each time with the following condition and the **.IndexOf(...)** method. This can be done with a loop by **fullWord** (leaving it to the reader for exercise), and it can be done in the lazy way by copy-paste.

```
if (word.IndexOf(pattern[i2]) == -1)
    word += pattern[i2];
// ...
```

The `.IndexOf(...)` method returns the index of the particular element if it is found or `-1` if the item is not found. Therefore, every time we get `-1`, it means that we still do not have this letter in the new string with unique letters and we can add it, and if we get a value other than `-1`, this will mean we already have the letter and we'll not add it.

Calculating Weight

Calculating the weight is simply going through the unique word (`word`) obtained in the last step, and for each letter we need to take its weight and multiply it by the position. For each letter, we need to calculate what value we will multiply its position by, for example by using a `switch` construction.

```
int multiplier = 0;
switch (word[i])
{
    case 'a':
        multiplier = 5;
        break;
    case 'b':
        multiplier = -12;
        break;
    case 'c':
        multiplier = 47;
        break;
    case 'd':
        multiplier = 7;
        break;
    case 'e':
        multiplier = -32;
        break;
    default:
        break;
}
```

Once we have the value of that letter, we should multiply it by its position. Because the indexes in the string differ by 1 from the actual positions, i.e. index 0 is position 1, index 1 is position 2, etc., we will add 1 to the indexes.

```
weight += multiplier * (i + 1);
```

All intermediate results obtained must be added to the total amount for each letter of the 5-letter combination.

Preparing the Output

Whether a word needs to be printed is determined by its weight. We need a condition to determine if the current weight is in the range `[start ... end]` passed to the input at the start of the program. If this is the case, we print the full word (`fullWord`).

Be careful not to print the word with unique letters. It was only needed to calculate the weight!

The words are separated with a space and we'll accumulate them in an intermediate variable `result`, which is defined as an empty string at the beginning.

```
if (weight >= firstNumber && weight <= secondNumber)
{
    result += fullWord + " ";
}
```

Final Touches

The condition is met unless we do not have a single word in the entered range. In order to find out if we have found a word, we can simply check whether the string `result` has its initial value (i.e., an empty string), if it does, we print **No**, otherwise we print the whole string without the last space (using the `.Trim()` method).

```
if (result == string.Empty)
{
    Console.WriteLine("No");
}
else
{
    Console.WriteLine(result.Trim());
}
```

Testing in the Judge System

Test your solution here: <https://judge.softuni.org/Contests/Practice/Index/518#2>.

Chapter 9.2. Problems for Champions – Part II

In this chapter we will review three additional problems that belong to the category "For Champions", i.e. they are more complex than the rest of the problems in this book.

More Complex Problems on the Studied Material

Before we move on to particular tasks, we must clarify that these can be solved in an easier way with **additional knowledge in programming with C#** (methods, arrays, collections, recursion, etc.), but each solution that will be provided now only uses the material covered in this book. The goal is to learn how to construct **more complex algorithms** based on your knowledge studied until now.

Problem: Passion Shopping Days

Lina has a real shopping passion. When she has some money, she immediately goes to the closest shopping center (mall) and tries to spend as much as she can on clothes, bags and shoes. But her favorite thing are winter sales. Our task is to analyze her strange behavior and **calculate the purchases** that Lina does when she enters the mall, as well as the **money she has left** when the shopping is over. All prices and money are in BGN (Bulgarian levs, lv).

The **first line** of the input will pass the **amount** that Lina has **before** she starts shopping. After that, upon reading the "**mall.Enter**" command, Lina enters the mall and starts shopping until the "**mall.Exit**" command is given. When Lina starts shopping, **on each line** of the input you will be given strings that are **actions performed by Lina**. Each **symbol** in the string is a **purchase or another action**. String commands contain only symbols of the **ASCII table**. The ASCII code of each sign is **related to what Lina must pay** for each of the goods. You need to interpret the symbols in the following way:

- If the symbol is a **capital letter**, Lina gets a **50% discount**, which means that you must decrease the money she has by 50% of the numeric representation of the symbol from the ASCII table.
- If the symbol is a **small letter**, Lina gets a **70% discount**, which means that you must decrease the money she has by 30% of the numeric representation of the symbol from the ASCII table.
- If the symbol is "%", Lina makes a **purchase** that decreases her money in half.
- If the symbol is "*", Lina **withdraws money from her debit card** and adds 10 lv. to her available funds.
- If the symbol is **different from all of the aforementioned**, Lina just makes a purchase without discount, and in this case you should simply subtract the value of the symbol from the ASCII table from her available funds.

If a certain value of her purchases is **higher** than her current available funds, Lina **DOES NOT** make the purchase. Lina's funds **cannot be less than 0**.

The shopping ends when the "**mall.Exit**" command is given. When this happens, you need to **print** the number of purchases made and the **money** that Lina has left.

Input Data

The input data must be read from the console. The **first line** of the input will indicate the **amount that Lina has before starting to purchase**. On each of the following lines there will be a particular command. After you read the command "**mall.Enter**", on each of the following lines you will be given strings holding **information regarding the purchases / actions** that Lina wants to perform. These strings will keep being passed, until the "**mall.Exit**" command is given.

Always only one "**mall.Enter**" command will be given, as well as only one "**mall.Exit**" command.

Output Data

The output data must be **printed on the console**. When shopping is over, you must print on the console a particular output depending on what purchases have been made.

- If no purchases have been made – "No purchases. Money left: {remaining funds} lv."
- If at least one purchase is made – "{number of purchases} purchases. Money left: {remaining funds} lv."

The funds must be printed with accuracy of up to 2 symbols after the decimal point.

Constraints

- Money is a **float** number within the range: $[0 - 7.9 \times 10^{28}]$.
- The count of strings between "**mall.Enter**" and "**mall.Exit**" will be within the range: **[1-20]**.
- The count of symbols in each string that represents a command will be within the range: **[1-20]**.
- Allowed execution time: **0.1 seconds**.
- Allowed memory: **16 MB**.

Sample Input and Output

Input	Output	Comments
110 mall.Enter d mall.Exit	1 purchases. Money left: 80.00 lv.	'd' has an ASCII code of 100. 'd' is a small letter, this is why Lina gets a 70% discount. She spends 30% of 100, which is 30 lv. After this purchase, she has: $110 - 30 = 80$ lv.

Input	Output	Input	Output
110 mall.Enter % mall.Exit	1 purchases. Money left: 55.00 lv.	100 mall.Enter Ab ** mall.Exit	2 purchases. Money left: 58.10 lv.

Hints and Guidelines

We will separate the solution of the problem into three main parts:

- Processing of the **input**.
- **Algorithm** for solving the problem.
- Formatting the **output**.

Let's examine each of the parts in details.

Processing the Input Data

The input of our task consists of a few components:

- On the **first line** we have all the **money** that Lina has for shopping.
- On **each of the following lines** we will have some kind of a **command**.

The first part of reading the input is trivial:

```
decimal shoppingMoney = decimal.Parse(Console.ReadLine());
```

But the second one contains a detail that we need to take into consideration. The requirements state the following:

On each of the following lines there will be a particular command. After you read the command "mall.Enter", on each of the following lines you will be given strings containing information regarding the purchases / actions that Lina wants to perform.

This is where we need to take into consideration the fact that from the **second input line on, we need to start reading commands**, but **only after we read the command "mall.Enter"**, we must start processing them. How can we do this? Using a **while** or a **do-while** loop is a good option. Here is an exemplary solution of how **to skip** all commands before processing the command "mall.Enter":

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

while (command != "mall.Enter")
{
    command = Console.ReadLine();
}

command = Console.ReadLine();
```

Here is the place to point out that calling `Console.ReadLine()` after the end of the loop is used for moving to the first command for processing.

Algorithm for Solving the Problem

The algorithm for solving the problem is a direct one – we continue **reading commands** from the console, **until the command "mall.Exit" is passed**. In the meantime, we **process** each symbol (**char**) of each one of the commands according to the rules specified in the task requirements, and in parallel, we **modify the amount** that Lina has, and **store the number of purchases**.

Let's examine the first two problems for our algorithm. The first problem concerns the way we read the commands until we reach the "mall.Exit" command. The solution that we previously saw uses a **while-loop**. The second problem for the task is to **process each symbol** of the command passed. Keeping in mind that the input data with the commands is **string** type, the easiest way to access each symbol inside the strings is via a **foreach loop**:

```
while (command != "mall.Exit")
{
    foreach (char action in command)
    {

    }

    command = Console.ReadLine();
}
```

Processing Command Symbols

The next part of the algorithm is to **process the symbols from the commands**, according to the following rules in the requirements:

- If the symbol is a **capital letter**, Lina gets a 50% discount, which means that you must decrease the money she has by 50% of the numeric representation of the symbol from the ASCII table.
- If the symbol is a **small letter**, Lina gets a 70% discount, which means that you must decrease the money she has by 30% of the numeric representation of the symbol from the ASCII table.
- If the symbol is "%", Lina makes a purchase that decreases her money in half.
- If the symbol is "**", Lina withdraws money from her debit card and adds 10 lv. to her available funds.
- If the symbol is **different from all of the aforementioned**, Lina just makes a purchase without discount, and in this case you should simply subtract the value of the symbol from the ASCII table from her available funds.

Let's examine the problems that we will be facing in the first condition. The first one is how to distinguish if a particular **symbol is a capital letter**. We can use one of the following ways:

- Keeping in mind the fact that the letters in the alphabet have a particular order, we can use the following condition `action >= 'A' && action <= 'Z'`, in order to check if our symbol is within the capital letters range.
- We can use the `char.ToUpper(...)` function.

The other problem is how **to skip a particular symbol**, if it is not an operation that requires more money than Lina has. This is doable using the `continue` construction.

An exemplary condition for the first part of the requirements looks like this:

```
if (action >= 'A' && action <= 'Z')
{
    decimal price = action * 0.5m;
    if (shoppingMoney < price)
    {
        continue;
    }

    shoppingMoney -= price;
    purchases++;
}
```

Note: the variable "`purchases`" is of `int` type, in which we store the number of all purchases.

We believe the reader should not have difficulties implementing all the other conditions because they are very similar to the first one.

Formatting the Output

In the end of our task we must **print** a particular **output**, depending on the following condition:

- If no purchases have been made – "No purchases. Money left: {remaining funds} lv."
- If at least one purchase is made – "{number of purchases} purchases. Money left: {remaining funds} lv."

The printing operations are trivial, as the only thing we need to take into consideration is that **the amount has to be printed with accuracy of up to 2 symbols after the decimal point**.

How can we do that? We will leave the answer to this question to the reader.

Testing in the Judge System

Test your solution here: <https://judge.softuni.org/Contests/Practice/Index/519#0>.

Problem: Numerical Expression

Bonny is an exceptionally powerful witch. As her natural power is not sufficient to successfully fight vampires and werewolves, she has started to master the power of Expressions. An expression is very hard to master, because the spell relies on the ability to **quickly solve mathematical expressions**.

In order to use an "Expression spell", the witch must know the result of a mathematical expression in advance. An **Expression spell** consists of a few simple mathematical expressions. Each mathematical expression can contain operators for **summing up, subtraction, multiplying** and/or **division**.

The expression is solved without considering the mathematical rules for calculating numerical expressions. This means that the priority is applied according to the sequence of the operators, and not the type of calculation that they do. The expression **can contain brackets**, as **everything inside the brackets is calculated first**. Every expression can contain multiple brackets, but no nested brackets:

- An expression containing (...(...)) is an **invalid one**.
- An expression containing (...)...(...) is a **valid one**.

Example

The expression

4 + 6 / 5 + (4 * 9 - 8) / 7 * 2

is solved in the following way:

```
4 + 6 / 5 + (4 * 9 - 8) / 7 * 2 =
10 / 5 + (4 * 9 - 8) / 7 * 2 =
2 + (4 * 9 - 8) / 7 * 2 =
2 + (36 - 8) / 7 * 2 =
2 + 28 / 7 * 2 =
30 / 7 * 2 =
4.285714285714286 * 2 =
8.57172857142571 =
8.57
```

Bonny is very pretty, but not as wise, so she will need our help to master the power of Expressions.

Input Data

The input data consists of a single text line, passed from the console. It contains a **mathematical expression for calculation**. The line **always ends with the "=" symbol**. The "=" symbol means **end of the mathematical expression**.

The input data is always valid and always in the described format. No need to validate it.

Output Data

The output data must be printed on the console. The output consists of one line: the **result** of the calculated mathematical expression, rounded up to the **second digit after the decimal point**.

Constraints

- The expressions will consist of **maximum 2500 symbols**.
- The numbers of each mathematical expression will be within the range [1 ... 9].
- The operators in the mathematical expressions will always be among + (summing up), - (subtraction), / (division) or * (multiplying).
- The result of the mathematical expression will be within the range [-100000.00 ... 100000.00].
- Allowed execution time: **0.1 seconds**.
- Allowed memory: **16 MB**.

Sample Input and Output

Input	Output	Input	Output
4+6/5+(4*9-8)/7*2=	8.57	3+(6/5)+(2*3/7)*7/2*(9/4+4*1)=	110.63

Hints and Guidelines

As usual, we will first read and process the input, after that we will solve the problem, and finally, we will print the result, formatted as required.

Reading the Input Data

The input data will consist of exactly one text line read from the console. Here we have **two ways** to process the input. The first way is by **reading the entire line using the Console.ReadLine() command** and accessing each symbol (**char**) of the line via a **foreach loop**. The second one is by **reading the input symbol by symbol using the Console.Read() command** and processing each symbol.

We will use the second option to solve the problem.

```
int symbol = Console.Read();
```

Creating Helper Variables

For the goals of our task we need two variables:

- One variable where we will store the **current output**.
- One variable where we will store the **current operator** of our expression.

```
decimal result = 0;
int expressionOperator = '+';
```

We will clarify two details regarding the aforementioned code. The first one is the use of **decimal** type for **storing the output of our expression** in order to avoid any problems with the accuracy pertaining to the **float** and **double** type. The second one is the default value of the operator – it is **+**, so that the very first number can be added to our output.

Defining the Program Structure

Now that we already have our starting variables, we must decide **what will be the main structure** of our program. By the requirements we understand that **each expression ends with =**, i.e. we must read and process symbols until we reach a **=**. This is followed by an accurately written **while loop**.

```
while (symbol != '=')
{
    symbol = Console.Read();
}
```

The next step is the processing of our **symbol** variable. We have 3 possible cases for it:

- If the symbol is a **start of a sub-expression placed in brackets** i.e. the found symbol is a **(**.
- If the symbol is a **digit between 0 and 9**. But how can we check this? How can we check if our symbol is a digit? We can use for assistance the **ASCII code** of the symbol, via which we can use the following formula: **[ASCII code of our symbol] - [ASCII code of the symbol 0] = [the digit that represents the symbol]**. If the result of this condition is between 0 and 9, then our symbol is really a number.
- If the symbol is an **operator**, i.e. it is **+, -, * or /**.

```
if (symbol == '(')
{
}
else if (0 <= symbol - '0' && symbol - '0' <= 9)
{
}
else if (symbol == '+' ||
          symbol == '-' ||
          symbol == '/' ||
          symbol == '*')
{
}
```

Implementing the Proposed Idea

Let's examine the actions that we need to undertake in the relevant cases that we defined:

- If our symbol is an **operator**, then the only thing we need to do is to **set a new value for the expressionOperator variable**.
- If our symbol is a **digit**, then we need to **change the current result of the expression depending on the current operator**, i.e. if **expressionOperator** is a **-**, then we must **decrease the result by the numerical representation of the current symbol**. We can get the numerical representation of the current symbol via the formula that we used upon checking the condition for this case (**[ASCII code of our symbol] - [the ASCII code of the symbol 0] = [the digit that represents the symbol]**)

This is a sample code, implemented the above idea:

```
else if (0 <= symbol - '0' && symbol - '0' <= 9)
{
    switch (expressionOperator)
    {
        case '+':
            result += symbol - '0';
            break;
```

```

        case '-':
            result -= symbol - '0';
            break;
        case '*':
            result *= symbol - '0';
            break;
        case '/':
            result /= symbol - '0';
            break;
    }
}

else if (symbol == '+' ||
symbol == '-' ||
symbol == '/' ||
symbol == '*')
{
    expressionOperator = symbol;
}

```

- If our symbol is a (, this indicates the **beginning of a sub-expression** (an expression in brackets). By definition, **the sub-expression must be calculated before modifying the result of the whole expression** (the actions in brackets are performed first). This means that we will have a local result for the sub-expression and a local operator.

```

if (symbol == '(')
{
    decimal innerResult = 0;
    int innerOperator = '+';
    symbol = Console.Read();
}

```

Calculating the Sub-Expression Value

After that, in order to **calculate the sub-expression value**, we will use the same methods that we used for calculating the main expression – we use a **while loop** to **read symbols** (until we reach an) symbol). Depending on whether the read symbol is a number or an operator, we modify the result of the sub-expression. The implementation of these operations is identical to the above described implementation for calculating expressions. We believe the reader will be able to easily handle it.

After finishing the result calculation for our sub-expression, we **modify the result of the whole expression** depending on the value of the **expressionOperator**.

```

switch (expressionOperator)
{
    case '+':
        result += innerResult;
        break;
    case '-':
        result -= innerResult;

```

```

        break;
    case '*':
        result *= innerResult;
        break;
    case '/':
        result /= innerResult;
        break;
}

```

Formatting the Output

The only output that the program must print on the console is the **result of solving the expression with accuracy of up to two symbols after the decimal point**. How can we format the output this way? We will leave the answer to this question to the reader.

Testing in the Judge System

Test your solution here: <https://judge.softuni.org/Contests/Practice/Index/519#1>.

Problem: Bulls and Cows

We all know the game called "Bulls and Cows": http://en.wikipedia.org/wiki/Bulls_and_cows. Upon having a particular 4-digit **secret number** and a 4-digit **suggested number**, the following rules are applied:

- If a digit in the suggested number matches a digit in the secret number and is located at the **same position**, we have a **bull**.
- If a digit in the suggested number matches a digit in the secret number, but is located at a **different position**, we have a **cow**.

Secret number	1	4	8	1	Comment
Suggested number	8	8	1	1	Bulls = 1 Cows = 2
Secret number	1	4	8	1	Comment
Suggested number	9	9	2	4	Bulls = 0 Cows = 2

Secret number	1	4	8	1	Comment
Suggested number	9	9	2	4	Bulls = 0 Cows = 2
Secret number	1	4	8	1	Comment
Suggested number	9	9	2	4	Bulls = 0 Cows = 2

Upon having a particular secret number and the bulls and cows pertaining to it, our task is **to find all possible suggested numbers** in ascending order.

If there are **no suggested numbers** that match the provided criteria provided, we must print "**No**".

Input Data

The input data is read from the console. The input consists of 3 text lines:

- The first line contains **the secret number**.
- The second line contains **the number of bulls**.
- The third line contains **the number of cows**.

The input data will always be valid. There is no need to verify them.

Output Data

The output data must be printed on the console. The output must consist of **a single line**, holding **all suggested numbers**, space separated. If there are **no suggested numbers** that match the criteria provided from the console, we must **print "No"**.

Constraints

- The secret number will always consist of **4 digits in the range [1..9]**.
- The number of **cows and bulls** will always be in the range **[0..9]**.
- Allowed execution time: **0.15 seconds**.
- Allowed memory: **16 MB**.

Sample Input and Output

Input	Output
2228 2 1	1222 2122 2212 2232 2242 2252 2262 2272 2281 2283 2284 2285 2286 2287 2289 2292 2322 2422 2522 2622 2722 2821 2823 2824 2825 2826 2827 2829 2922 3222 4222 5222 6222 7222 8221 8223 8224 8225 8226 8227 8229 9222

Input	Output
1234 3 0	1134 1214 1224 1231 1232 1233 1235 1236 1237 1238 1239 1244 1254 1264 1274 1284 1294 1334 1434 1534 1634 1734 1834 1934 2234 3234 4234 5234 6234 7234 8234 9234

Hints and Guidelines

We will solve the problem in a few steps:

- We will read the **input data**.
- We will generate all possible **four-digit combinations** (candidates for verification).
- For each generated combination we will calculate **how many bulls** and **how many cows** it has according to the secret number. Upon matching the needed bulls and cows, we will **print the combination**.

Reading the Input Data

We have 3 lines in the input data:

- **Secret number**.
- **Number of desired bulls**.
- **Number of desired cows**.

Reading the input data is trivial:

```
int guessNumber = int.Parse(Console.ReadLine());
int targetBulls = int.Parse(Console.ReadLine());
int targetCows = int.Parse(Console.ReadLine());
```

Declaring a Flag

Before starting to write the algorithm for solving our problem, we must **declare a flag** that indicates whether a solution is found:

```
bool solutionFound = false;
```

If after finishing our algorithm this flag is still **false**, then we will print **No** on the console, as specified in the requirements.

```
if (!solutionFound)
{
    Console.WriteLine("No");
}
```

Generating Four-Digit Numbers

Let's start analyzing our problem. What we need to do is **analyze all numbers from 1111 to 9999**, excluding those that contain zeroes (for example **9011**, **3401**, etc. are invalid). What is the easiest way to **generate** all these **numbers**? We will **use nested loops**. As we have a **4-digit number**, we will have **4 nested loops**, as each of them will generate **an individual digit** in our number for testing.

```
for (int digit1 = 1; digit1 <= 9; digit1++)
{
    for (int digit2 = 1; digit2 <= 9; digit2++)
    {
        for (int digit3 = 1; digit3 <= 9; digit3++)
        {
            for (int digit4 = 1; digit4 <= 9; digit4++)
            {

```

Thanks to these loops, we have access to every **digit** of all numbers that we need to check. Our next step is to separate the secret number into digits. This can be achieved very easily using **a combination of integer division and modular division**.

```
int guessDigit1 = (guessNumber / 1000) % 10;
int guessDigit2 = (guessNumber / 100) % 10;
int guessDigit3 = (guessNumber / 10) % 10;
int guessDigit4 = (guessNumber / 1) % 10;
```

Creating Additional Variables

Only two last steps remain until we start analyzing how many cows and bulls there are in a particular number. Accordingly, the first one is the **declaration of counter variables** in the nested loops, in order to **count the cows and bulls** for the current number. The second step is to make **copies of the digits of the current number** that we will analyze, in order to prevent problems upon working with nested loops, in case we make changes to them.

```
int digitToCheck1 = digit1;
int digitToCheck2 = digit2;
int digitToCheck3 = digit3;
int digitToCheck4 = digit4;

int currentBulls = 0;
int currentCows = 0;
```

We are ready to start analyzing the generated numbers.

Counting the Bulls

What logic can we use? The easiest way to check how many cows and bulls there are inside a number is via a **sequence of if-else conditions**. Yes, this is not the most optimal way, but in order to stick to what is covered in the current book, we will use this approach.

What conditions do we need?

The condition for the bulls is very simple – we check whether the **first digit** of the generated number matches the **same digit** in the secret number. We remove the digits that are already checked in order to avoid repetitions of bulls and cows.

```
// Find all bulls, count them and remove them (assign -1 and -2)
if (digitToCheck1 == guessDigit1)
{
    // Bull at position #1 found -> count it and remove it
    currentBulls++;
    guessDigit1 = -1;
    digitToCheck1 = -2;
}
```

We repeat the action for the second, third and fourth digit.

Counting the Cows

We will apply the following condition for the cows – first we will check whether the **first digit** of the generated number **matches the second one**, the **third one** or the **fourth digit** of the secret number. An example for the implementation:

```
// Find all cows for digitToCheck1, count them and remove them (assign -1)
if (digitToCheck1 == guessDigit2)
{
    //Cow at position #2 found -> count it and remove it
    currentCows++;
    guessDigit2 = -1;
}
else if (digitToCheck1 == guessDigit3)
{
    // Cow at position #3 found -> count it and remove it
    currentCows++;
    guessDigit3 = -1;
}
else if (digitToCheck1 == guessDigit4)
{
    // Cow at position #4 found -> count it and remove it
    currentCows++;
    guessDigit4 = -1;
}
```

After that, we sequentially check whether the **second digit** of the generated number **matches the first one**, the **third one** or the **fourth digit** of the secret number; whether the **third digit** of the generated

number matches the **first one**, the **second one** or the **fourth digit** of the secret number; and finally, we check whether the **fourth digit** of the generated number matches the **first one**, the **second one** or the **third digit** of the secret number.

Printing the Output

After completing all conditions, we just need to check whether the bulls and cows in the currently generated number match the desired bulls and cows read from the console. If this is true, we print the current number on the console.

```
if (currentBulls == targetBulls && currentCows == targetCows)
{
    if (solutionFound)
    {
        Console.Write(" ");
    }

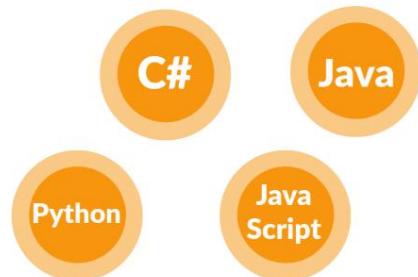
    Console.WriteLine($"{digit1}{digit2}{digit3}{digit4}");
    solutionFound = true;
}
```

Testing in the Judge System

Test your solution here: <https://judge.softuni.org/Contests/Practice/Index/519#2>.

High-quality education profession and job for software engineers

- ✓ Free start for **absolute beginners**
- ✓ Choose among the **most in-demand** professions in the software industry
- ✓ Assistance for **career start**
- ✓ **Top trainers**, mentors and active learning **community**



SoftUni provides practical, modern and **innovative education** for programming, IT, design, marketing and digital skills for thousands of young people. The "**Software University**" program builds true **software engineering professionals**.

Join the end-to-end **software engineering learning program** at SoftUni to master the most in-demand programming concepts, languages, software technologies and skills through a modern teaching methodology, **interactive learning platform** and tons of practical exercises and projects. Our **curriculum** is developed together with the **IT companies** to closely match the industry demands.

SoftUni works directly with the **companies** from the software industry to **find a job** for its students and to make them successful software engineers.

The path of the student at SoftUni



Apply now

softuni.org/apply

Chapter 10. Methods

In the current chapter we get familiar with **methods** and learn what they **are**, and which are the **base concepts** when working with them. We will also learn why it is a **good practice** to use them, how to declare and call them. We will look at **parameters** and **return value of a method**, and also understand how to use the returned value. At the end of the chapter we will look at **the established practices** when using methods.

Introduction by Examples

Methods allow invoking a parameterized **named piece of code** several times from many places in the code. Example:

```
static void PrintLetter(char letter, int count)
{
    for (int i = 0; i < count; i++)
        Console.Write(letter + " ");
    Console.WriteLine();
}

static void Main()
{
    int count = 0;
    for (char letter = 'a'; letter <= 'd'; letter++)
        PrintLetter(letter, ++count);

    for (char letter = 'c'; letter >= 'a'; letter--)
        PrintLetter(letter, --count);
}
```

Run the above code example: <https://repl.it/@nakov/triangle-of-letters-csharp>.

The above code produces the following **output**:

```
a
b b
c c c
d d d d
c c c
b b
a
```

The above **method** (named piece of code) **PrintLetter(letter, count)** prints given character (**letter**) several times (**count**). It is invoked several times in two loops from the program **Main()** method.

Let's get into details how methods are **defined**, how methods are **invoked**, how methods accept **parameters** (input values) and how methods return **results** (output values).

Methods can take parameters and can **return values**, e.g.

```
static double CalcCircleArea(double radius)
{
    return Math.PI * radius * radius;
}
```

```
static void Main()
{
    Console.WriteLine("r = {0}, area = {1}", 5, CalcCircleArea(5));
    Console.WriteLine("r = {0}, area = {1}", 2.8, CalcCircleArea(2.8));
}
```

Run the above code example: <https://repl.it/@nakov/circle-area-methods-csharp>.

The output from the above code is like this:

```
r = 5, area = 78.5398163397448
r = 2.8, area = 24.630086404144
```

Let's get into details on how to **define**, **invoke** and **use methods** in C#, how to take and pass **parameters** and to **return values**.

What Is a "Method"?

Up until now we found out that when **writing** the code of a program that solves a problem, it is **easier** to **divide** the task into **parts**. Each part fulfills a **given action** and this way it is not only **easier** to solve the task, but the **readability** of the code and checking for mistakes is significantly better.

Each piece of code that executes some functionality and has been separated logically can take the functionality of the method. This is exactly what **methods are – pieces of code with names** given by us in a certain way, which can be **called** as many times as we need them.

A method can be called as many times, as we think we need, in order to solve a problem. This **saves** us repeating the same code over and over again, and also **reduces** the possibility to make a mistake when correcting the code.

Simple Methods

Simple methods are used for performing a certain **action** that **helps** to solve a given problem. These actions can be printing a string on the console, doing a verification, executing a loop, etc.

Let's see the following example of a **simple method**:

```
static void PrintHeader()
{
    Console.WriteLine("-----");
}
```

This method prints a header, which is a sequence of the `-` symbol. Because of this, its name is **PrintHeader**. The parentheses `(` and `)` **always** follow the name, no matter what the method is called. We will later see how to name the methods we work with, but for now, we will only say that it is important for **its name to describe the action** that the method is doing.

The **body** of the method contains **the program code**, which is between the curly brackets `{` and `}`. These brackets **always** follow its **declaration** and between them we write the code, which solves the problem described by the method's name.

To **call this method**, we just write its **name**, along with `()` like it is shown below:

```
PrintHeader();
```

A method should be **called from a code inside another method**, e.g. from the **Main()** method of the C# program:

```
static void Main()
{
    PrintHeader();
}
```

Why Use Methods?

So far we determined that methods help with **dividing larger programs into smaller parts**, which leads to **easier solving** of the problem in question. This makes our program not only better structured and easier to read, but more understandable as well.

By using methods, we **avoid repeating** code. **Repetition** is **bad** practice, because it **complicates maintaining** the program and leads to errors. If a certain part of our code can be found more than once in the program and we need to change it, the changes must be made in all of the repetitions of the code in question. There is a great probability to miss a spot where correction is needed, which would lead to incorrect behavior of the program. This is the reason why it is a **good practice** to use a certain fragment of code **more than once** in our program, to **define it as a separate method**.

Methods make it **possible** to use certain **code multiple** times. With solving more and more problems, you will find that using already existing methods saves a lot of time and effort.

Declaring Methods

In C# you can **declare** methods inside a class, i.e. between the opening **{** and closing **}** brackets of the class. Declaring is registering the method in the program, so that it can be recognized in the rest of it. The best-known example is the **Main(...)** method, which we use in every program that we write.

```
class Methods
{
    0 references
    static void Main()
    {
    }
}
```

With the next example we will look at the mandatory elements in the declaration of a method.

```
static double GetSquare(double num)
{
    return num * num;
}
```

- **Type of the returned value.** In this case the type is **double**, which means that the method will **return a result**, which is of **double** type. The returned value can be **int**, **double**, **string** etc., and also **void**. If the type is **void**, this means that the method **doesn't return** a result, but only **does a particular operation**.
- **Method name.** The name of the method is **defined by us**, but we shouldn't forget that it has to **describe the function**, which is executed by the code in its body. In the example the name is **GetSquare**, which tells us that this method is made to find the area of a square.

- **Parameters list.** It is declared between the parentheses (and) that we write after its name. This is where we list all the **parameters** that the method will use. There can be **only one** parameter, **multiple** ones or it could be an **empty** list. If there aren't any parameters, we will write only the parentheses (). In this example we declare the parameter **double num**.
- **static** declaration in the method description. For the moment you can accept that we write **static** always when we declare a method, and later when we get familiar with object-oriented programming (OOP), we will learn about the difference between **static methods** (shared for the whole class) and **methods of an object**, which work on the data of a certain instance of the class (object).

When declaring methods, you must follow the **sequence** of its base elements – first **type of the returned value**, then the **method name** and in the end the **list of parameters**, surrounded by parentheses () .

After we have declared a method, its **implementation (body)** follows. In the body of the method we write down **the algorithm**, by which it solves a problem, i.e. the body contains the code (program block), which applies the method's **logic**. In the shown example we are calculating the area of a square, which is **num * num**.

When declaring a variable in the body of a method, it is called a **local** variable of the method. The area where this variable exists and can be used starts from the line where we have declared it and reaches the closing curly bracket } of the body of the method. This area is called **variable scope**.

Calling Methods

Calling a method means **starting to execute the code**, which is in **the body of the method**. This happens by writing its **name**, followed by parentheses () and the semicolon sign ; to end the line. If the method needs input data, it is given in the parentheses (), and the succession of the parameters should be the same as the one of the given parameters when declaring the method. Here is an example:

```
// Declaring method
static void PrintHeader()
{
    Console.WriteLine("-----");
}
static void Main()
{
    // Invoking the declared method
    PrintHeader();
}
```

A method can be called from **several places** in the program. One way is to call it in **the main method**.

```
static void Main()
{
    // Invoking the declared method
    // from the Main() method body
    PrintHeader();
}
```

A method can also be called from **the body of another method**, which is **not** the main method of the program.

```
static void PrintHeader()
{
    // Invoking methods from another method
    PrintHeaderTop();
    PrintHeaderBottom();
}
```

There is also a possibility for the method to be called in **its own body**. This is called **recursion** and you can find more information about it in Wikipedia (<https://en.wikipedia.org/wiki/Recursion>) or you can search on your own in the Internet.

It is important to know that if a method is declared in a class, it can be called before the line, on which it has been declared.

Example: Blank Receipt

Write a method that prints a blank receipt. The method should call another three methods: one to print the header, one for the middle part of the receipt and one for the lower part.

Part of the receipt	Text
Upper part	CASH RECEIPT -----
Middle part	Charged to_____ Received by_____
Lower part	----- (c) SoftUni

Sample Input and Output

Input	Output
(no input)	CASH RECEIPT ----- Charged to_____ Received by_____ ----- (c) SoftUni

Hints and Guidelines

The first step is to create a **void** method to **print the upper part** of the receipt (header). Let's give it a meaningful name, which describes what the method does, e.g. **PrintReceiptHeader**. In its body write the code from the example below:

```
static void PrintReceiptHeader()
{
    Console.WriteLine("CASH RECEIPT");
    Console.WriteLine("-----");
}
```

In the same way we'll create two more methods **to print the middle part** of the receipt (body) **PrintReceiptBody** and **to print the lower part** of the receipt (footer) **PrintReceiptFooter**.

After this we will create **another method**, which will call the three methods we already wrote, one after the other:

```
static void PrintReceipt()
{
    PrintReceiptHeader();
    PrintReceiptBody();
    PrintReceiptFooter();
}
```

In the end we'll **call** the **PrintReceipt** method from the body of the **Main** method of our program:

```
static void Main()
{
    PrintReceipt();
}
```

Testing in the Judge System

The program with five methods that are invoked from one another is ready and we can **run and test** it, after which we can send it for **automated evaluation** and grading in the SoftUni judge system: <https://judge.softuni.org/Contests/Practice/Index/594#0>.

Methods with Parameters

Frequently in order to solve a problem, the method by which we do this needs **additional information**, which depends on its purpose. This is precisely the information that **the method parameters** are, and its behavior depends on them.

Using Parameters in Methods

As we observed above, **the parameters can be zero, one or more**. When declaring them you should divide them with a comma. They can be of any type (**int**, **string** etc.), and there is an example below to show how they can be used by the method.

We **declare** the method and its **list of parameters**, then we write the code that the method executes.

```
static void PrintNumbers(int start, int end)
{
    for (int i = start; i <= end; i++)
    {
        Console.WriteLine("{0} ", i);
    }
}
```

After that **call** the method and **give it particular values**:

```
static void Main()
{
    PrintNumbers(5, 10);
}
```

When **declaring parameters**, we can use **various** types of variables, and we should be careful that every parameter has a **type** and **name**. It is important to note that when calling the method, we must pass to it **values** for the parameters in **the order**, in which they are **declared**. If the parameters are first **int** and after that **string**, when calling the method, we can't give it first a **string** and then **int**. We can only change places of given parameters if we write the name of the parameter beforehand, as you will see below in one of the examples. This is generally not a good practice!

Let's look at the example for declaring a method, which has several parameters of different types.

```
static void PrintStudent(string name, int age, double grade)
{
    Console.WriteLine("Student: {0}; Age: {1}, Grade: {2}",
        name, age, grade);
}
```

Example: Sign of an Integer

Create a method that prints the **sign** of an integer **n**.

Sample Input and Output

Input	Output
2	The number 2 is positive.
-5	The number -5 is negative.
0	The number 0 is zero.

Hints and Guidelines

The first step is **creating** a method and giving it a descriptive name, e.g. **PrintSign**. This method will have only one parameter of **int** type.

```
static void PrintSign(int n)
{
}
```

The next step is **implementing** the logic by which the program will check what the sign of the number is. You can see from the examples that there are three cases – the number is larger than, equal to or lower than zero, which means that we'll make three verifications in our method.

The next step is to read the input number and to call the new method from the body of the **Main** method.

```
static void Main()
{
    int n = int.Parse(Console.ReadLine());
    PrintSign(n);
}
```

Testing in the Judge System

Test your solution here: <https://judge.softuni.org/Contests/Practice/Index/594#1>.

Optional Parameters

C# supports using **optional** parameters. They allow **missing** parameters when calling the method. Their declaring is done by **providing default value** in the description of the parameter.

The following example illustrates using optional parameters:

```
static void PrintNumbers(int start = 0, int end = 100)
{
    for (int i = start; i <= end; i++)
        Console.WriteLine("{0} ", i);
}
```

The shown method **PrintNumbers** can be called in one of several ways:

```
static void Main()
{
    PrintNumbers(5, 10);
    PrintNumbers(15);
    PrintNumbers();
    PrintNumbers(end: 40, start: 35);
}
```

Example: Printing a Triangle

Create a method which prints a triangle as in the examples.

Sample Input and Output

Input	Output	Input	Output	Input	Output	Input	Output
2	1 1 2 1	3	1 1 2 1 2 3 1 2 1	4	1 1 2 1 2 3 1 2 3 4 1 2 3 1 2 1	1	1

Hints and Guidelines

Before creating a method to print a row with a given beginning and an end, we must read the input number from the console. After that we choose a meaningful name, which describes its purpose, e.g. **PrintLine**, and implement it.

```
static void PrintLine(int start, int end)
{
    for (int i = start; i <= end; i++)
    {
        Console.Write(i + " ");
    }
    Console.WriteLine();
}
```

From the drawing exercises we remember that it is good practice to divide the figure into several parts. To make it easier we will divide the triangle into three parts – upper, middle and lower.

The next step is to print the upper half of the triangle using a loop:

```
for (int i = 0; i < n; i++)
{
    PrintLine(1, i);
}
```

Then we print the middle part:

```
PrintLine(1, n);
```

In the end we print the lower part of the triangle, but this time the loop step decreases.

```
for (int i = n - 1; i > 0; i--)
{
    PrintLine(1, i);
}
```

Testing in the Judge System

Test your solution here: <https://judge.softuni.org/Contests/Practice/Index/594#2>.

Example: Draw a Filled Square

Print on the console a filled square with side n, as in the examples below.

Sample Input and Output

Input	Output	Input	Output	Input	Output
4	----- - \ / \ / - - \ / \ / - -----	3	----- - \ / \ / - - \ / \ / - -----	2	-----

Hints and Guidelines

The first step is to read the input from the console. After that we need to create a method, which will print the first and the last rows because they are the same. Let's remember that we must give it a descriptive name and give it as a parameter the length of the side. We will use the constructor `new string`.

```
static void PrintHeaderFooter(int n)
{
    Console.WriteLine(new string('-', 2 * n));
}
```

Our next step is to create a method that will draw the middle rows on the console. Again, give it a descriptive name i.e. `PrintMiddleRow`. This is a sample code:

```
static void PrintMiddleRow(int n)
{
    Console.Write("-");
```

```

for (int i = 0; i < n - 1; i++)
{
    Console.Write("\\\\");
}
Console.Write("-");
Console.WriteLine();
}

```

Finally, call the methods in the `Main()` method of the program in order to draw the whole square:

```

static void Main()
{
    int input = int.Parse(Console.ReadLine());
    PrintHeaderFooter(input);
    // TODO: Draw the rest of the square
}

```

Testing in the Judge System

Test your solution here: <https://judge.softuni.org/Contests/Practice/Index/594#3>.

Returning a Result from a Method

We already examined methods that do a specific task, for example printing some text, a number or a figure on the console. There are also other types of methods that can **return** some kind of **result**. These are the methods we are going to analyze in the next lines.

Types of Returned Values

Up until now we saw some examples, in which when declaring methods, we used the keyword `void`, which shows that the method **does not** return a result, but just executes a certain action.

```

static void AddOne(int n)
{
    n += 1;
    Console.WriteLine(n);
}

```

If we **replace void by a type** of some variable, this will tell the program that the method should return a value of the said type. This returned value could be of any type – `int`, `string`, `double` etc.



In order for a method to return **a result** we need to write the type of returned value we expect when declaring the method, in the place of `void`.

```

static int PlusOne(int n)
{
    return n + 1;
}

```

We should note that **the result** returned by the method can be of **a type, compatible with the type of the returned value** of the method. For example, if the declared type of the returned value is `double`, we can return a value of `int` type.

The "Return" Operator

In order to obtain a result from the method we need to use the **return** operator. It should be **used in the body** of the method and tells the program to **stop its execution** and to **return** the method invoker a certain **value**, which is defined by the expression after the **return** operator.

In the example below there is a method that reads two names from the console, concatenates them and returns them as a result. The return value is of **string** type:

```
static string ReadFullName()
{
    string firstName = Console.ReadLine();
    string lastName = Console.ReadLine();
    return firstName + " " + lastName;
}
```

The **return** operator can also be used in **void** methods. This way the method will stop its execution without returning a value, and after it there shouldn't be an expression, which should be returned. In this case we use **return** only to exit the method.

There are cases where **return** can be called from multiple places in the method, but only if there are certain input conditions.

The "Return" Operator – Example

We have a method in the example below, which **compares two numbers** and returns a result respectively **-1, 0 or 1** depending on if the first argument is smaller, equal or larger than the second argument, given to the function. The method uses the keyword **return** in three different places, in order to return three different values according to the logic of comparing the numbers:

```
static int CompareTo(int number1, int number2)
{
    if (number1 > number2)
    {
        return 1;
    }
    else if (number1 == number2)
    {
        return 0;
    }
    else
    {
        return -1;
    }
}
```

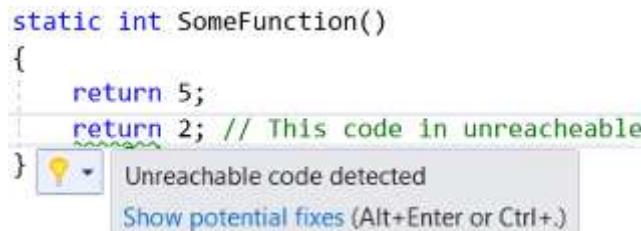
We can **invoke** the above method and ensure that it behaves as expected:

- **CompareTo(3, 4) → -1**
- **CompareTo(20, 10) → 1**
- **CompareTo(5, 5) → 0**
- **CompareTo(-5, -7) → 1**
- **CompareTo(-10, -5) → -1**

The Code After "Return" is Inaccessible

After the `return` operator, there **should not** be any more lines of code in the current block, because if there are, Visual Studio will warn you that it has found a piece of code **that is inaccessible**:

```
static int SomeFunction()
{
    return 5;
    return 2; // This code is unreachable
}
```



A screenshot of Visual Studio code editor showing a warning for unreachable code. The line `return 2;` is underlined with a green squiggle, and a tooltip box appears with the text "Unreachable code detected" and "Show potential fixes (Alt+Enter or Ctrl+.)".



In programming you **can't have the `return` operator twice**, one after the other (double `return`), because executing the first one won't allow the execution of the second one. From time to time programmers joke by saying "`write return; return; and let's go`", in order to explain that the program logic is lost.

Using the Returned Value

After a method is executed and returns a value, this value can be used in **multiple** ways.

The first one is **to assign the result as a value of a variable** of a compatible type:

```
int max = GetMax(5, 10);
```

The second one is for the result to be used **in an expression**:

```
decimal total = GetPrice() * quantity * 1.20m;
```

The third one is to **pass** the result of the method to another method:

```
int age = int.Parse(Console.ReadLine());
```

Example: Calculating Triangle Area

Create a method that calculates the area of a triangle using the given base and height and returns it as a result.

Sample Input and Output

Input	Output
3	
4	6

Hints and Guidelines

The first step is to read the input. After that, **create** a method, but this time be careful when **declaring** to give it the correct **type** of data we want the method to return, which is **double**.

```
static double GetTriangleArea(double length,
    double height)
{
    return (length * height) / 2;
}
```

The next step is to call the `new` method from the `Main()` method and to store the returned value in a suitable variable.

```
static void Main()
{
    double a = double.Parse(Console.ReadLine());
    double h = double.Parse(Console.ReadLine());
    double area = GetTriangleArea(a, h);
    Console.WriteLine(area);
}
```

Testing in the Judge System

Test your solution here: <https://judge.softuni.org/Contests/Practice/Index/594#4>.

Example: Math Power

Create a method that calculates and returns as result the power of a given number.

Sample Input and Output

Input	Output	Input	Output
2	256	3	81
8		4	

Hints and Guidelines

Our first step is to read the input data from the console. The next step is to create a method that will take two parameters (the number and the power) and returns as a result a number of `double` type.

```
static double CalculatePower(double number, double power)
{
    double result = 0d;
    // TODO: Calculate result
    // Use a loop or Math.Pow(...)
    return result;
}
```

After we have done the calculations, we have to only print the result in the `Main()` method of the program.

Testing in the Judge System

Test your solution here: <https://judge.softuni.org/Contests/Practice/Index/594#5>.

Methods Returning Multiple Values

There are cases in practice when we need a method to return more than one element as a result. For this to be possible `ValueTuple` has been integrated in Visual Studio and C# (C# 7 and later versions), as well as a literal of `ValueTuple` type. The `ValueTuple` type represents two values, which allow the temporary containing of `multiple values`. The values are contained in variables (fields – we will learn about them later) of the corresponding types. Although the type `Tuple` existed before C# 7, it didn't

have good support in the older versions and it's ineffective. That's why in previous versions of C# the elements in one **Tuple** were shown as **Item1**, **Item2** etc. and the names of their variables (the variables in which they are contained) could not be changed. In C# 7 the type (**ValueTuple**) is maintained, which allows giving meaningful names to the elements in a **ValueTuple**.

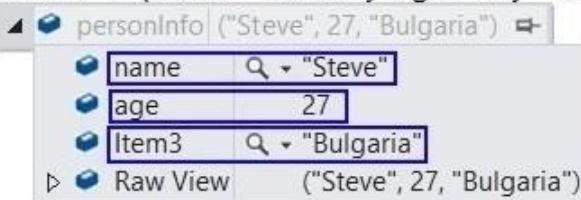
Declaring a ValueTuple

Let's examine an example declaring of a variable of **ValueTuple** type:

```
var personInfo = (name: "Steve", age: 27, "Bulgaria");
```

To make it easier when declaring, we use the keyword **var**, and in the brackets we list **the names of the values we want**, followed by **the values themselves**. Let's see what the variable **personInfo** contains in debug mode:

```
var personInfo = (name: "Steve", age: 27, "Bulgaria");
```



We can see that it contains several **fields with names and values**, which were given when initializing the variable. We can see that the last variable is called **Item3**. This is so because when initializing we haven't named the variable, which contains the value "**Bulgaria**". In this case the naming is **by default**, i.e. the variables are named **Item1**, **Item2**, **Item3**, etc.

Method Returning Multiple Values

The following method takes as parameters two integers (**x** and **y**) and **returns two values** – the result of integer division and the remainder:

```
static (int result, int remainder) Divide(int x, int y)
{
    int result = x / y;
    int remainder = x % y;
    return (result, remainder);
}
```

This method returns a result of **ValueTuple** type, containing two variables (fields) of **int** type, named **result** and **remainder** respectively. Calling the method is done in the following way:

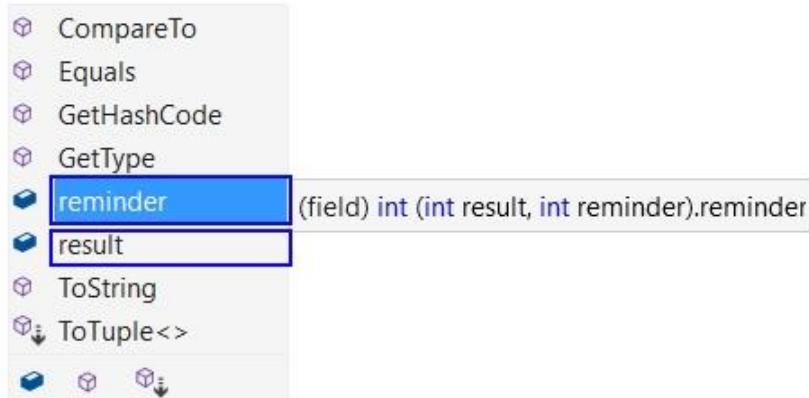
```
var division = Divide(1, 3);
```

In order to get the results returned from the method, we apply **point notation to the variable **division****, as it is shown on the example below:

```
var division = Divide(1, 3);
int res = division.result;
int rem = division.reminder;
```

To simplify the developers, Visual Studio implements **auto-complete for tuples**, returned from a method, just as it can be expected:

division.



Method Overloading

In many programming languages like C# and Java the same method can be declared in **few variants** with the **same name and different parameters**. This goes by the term "**method overloading**". Now let's see how to write these overloaded methods in C#.

Method Signature

In programming methods are **identified** through the elements of their declaration: **name** of the method + a list of its **parameters**. These two elements define its **specification**, the so called "**method signature**".

The **method signature** is defined by the **method name** and the **definitions of the method parameters** (only parameter types are considered, and the parameter names are ignored). Example:

```
static void Print(string text)
{
    Console.WriteLine(text);
}
```

In this example the method's signature is its name (**Print**), together with its parameter types (**string**).

If our program holds several **methods with the same name**, but with **different lists of parameters (signatures)**, we can say that we use "**method overloading**".

Overloading Methods in C# Programs

As we mentioned, if you use **the same name for several methods with different signatures**, this means that you are **overloading a method**. The code below shows how three different methods can use the same name with different combinations of parameters and execute different actions.

```
static void Print(string text)
{
    Console.WriteLine(text);
}

static void Print(int number)
{
    Console.WriteLine(number);
}
```

```
static void Print(string text, int number)
{
    Console.WriteLine(text + " " + number);
}
```

Signature and Return Value Type

It is important to say that **the returned type as a result** of the method **is not a part of its signature**. If the returned type was a part of the signature, then the compiler doesn't know which method exactly to call (there is an ambiguity).

Let's look at the following **example**: we have two methods with different return types. Despite that, Visual Studio shows that there is a mistake, because both of their signatures are the same. Therefore, when trying to call a method named **Print(...)**, the compiler can't know which of the two methods to invoke.

```
static void Print(string text)
{
    Console.WriteLine(text);
}

static string Print(string text)
{
    return text;
}
```

Example: Greater of Two Values

The input is two values of the same type. The values can be of **int**, **char** or **string** type. Create a method **GetMax()** that returns as a result the greater of the two values.

Sample Input and Output

Input	Output	Input	Output	Input	Output
int 2 16		char a z		string Ivan Tod	

Creating the Methods

We need to create three methods with the same name and different signatures. First we create a method, which will **compare integers**.

```
static int GetMax(int first, int second)
{
    if (first >= second)
    {
        // TODO: return value
    }
    // TODO: handle other cases
}
```

Following the logic of the previous method we create another one with the same name, but this one will compare characters.

```
static char GetMax(char first, char second)
{
    // TODO: create logic
}
```

The next method we need to create will compare strings. The logic here is a bit different from the previous two methods because variables of **string** type cannot be compared with the operators **<** and **>**. We will use the method **CompareTo(...)**, which returns a numerical value: larger than 0 (the compared object is larger), smaller than 0 (the compared object is smaller) and 0 (the two objects are the same).

```
static string GetMax(string first, string second)
{
    if (first.CompareTo(second) >= 0)
    {
        // TODO: return value
    }
    // TODO: return value
}
```

Reading the Input Data and Using the Methods

The last step is to read the input data, to use the appropriate variables and to invoke the method **GetMax()** from the body of the **Main()** method.

```
static void Main()
{
    var type = Console.ReadLine();
    if (type == "int")
    {
        int first = int.Parse(Console.ReadLine());
        int second = int.Parse(Console.ReadLine());
        int max = GetMax(first, second);
        Console.WriteLine(max);
    }
    else if (type == "char")
    {
        // TODO: call GetMax() with char arguments
    }
    else if (type == "string")
    {
        // TODO: call GetMax() with string arguments
    }
}
```

Testing in the Judge System

Test your solution here: <https://judge.softuni.org/Contests/Practice/Index/594#6>.

Nested Methods (Local Functions)

Let's examine the following example:

```
static void Main()
{
    double first = 1.22;
    double second = 3.27;
    double Result(double a, double b)
    {
        return a + b;
    }
    Console.WriteLine(Result(first, second));
}
```

What Is a Local Function?

We can see that in this code the `Main()` method has another declared method `Result()`. This **nested** method is called **local function**.

Local functions can be declared inside every other method. When the C# compiler compiles such functions, they are turned into private methods. Because you will learn about the difference between **public** and **private** methods later, we can now say that **private** can be used only in the class, in which they have been declared. The programs that we write at this level are using only one class, therefore we can say that we can use nested methods without any concerns.

Why Use Local Functions?

With time and practice you will see that when you are writing code, you often need **methods that you only need once**, or the method you need gets very long. We said earlier that when a method contains too many lines of code it becomes hard to read and maintain.

This is where local functions come in handy – they help us **declare a new method in another one** we already have, and it will be used only once. This helps making our code better ordered and **easier to read**, which helps for faster correction if there is an error in the code and limits the possibility for mistakes when making changes in the program logic.

Declaring Local Functions

Let's look again at the previously mentioned example:

```
static void Main()
{
    double first = 1.22;
    double second = 3.27;
    double Result(double a, double b)
    {
        return a + b;
    }
    Console.WriteLine(Result(first, second));
}
```

In this example the `Result()` method is a **local function**, because it is nested in the `Main()` method, i.e. `Result()` is local for `Main()`. This means that the `Result()` method can only be used in the `Main()` method, because it's declared inside it. The only difference between nested and normal methods is that nested methods can't be **static**. Because the definition for **static** will be seen later, we will say for the moment that when declaring a local function, we write only the return value type, the name of the method and its list of parameters. In this case this is `double Result(double a, double b)`.

Local functions can access variables, which are in the method containing them. The next example demonstrates how this is happening:

```
static void Main()
{
    string output = "I am a local function";

    void PrintOutput()
    {
        Console.WriteLine(output);
    }
}
```

This feature of nested methods makes them very helpful when solving a problem. They save time and code, which we would otherwise lose to give them parameters and variables, which we can already use in nested methods.

Naming Methods

When you name methods, use **meaningful names**.

- Because every method **handles** a part of our problem, when naming it we should keep in mind **the action it does**, i.e. it is a good practice for **the name to describe what the method does**.
- Each method must do **only one task** and its name should describe this task. This principle in programming is known as "**strong cohesion**".
- In C# the method name must start with **uppercase letter** and should be made of a **verb** or a couple: **verb + noun**. The name is formatted by following the **Upper Camel Case** convention (PascalCase), i.e. **each word, including the first one, starts with uppercase**.
- The brackets (and) always follow the name (without spaces).

A few examples for **correctly named methods**:

- `FindStudent`, `LoadReport`, `Sine`

A few examples for **incorrectly named methods** (think why):

- `Method1`, `DoSomething`, `HandleStuff`, `SampleMethod`, `DirtyHack`

If you cannot think of an appropriate name, then the method most probably solves more than one task or doesn't have a **clearly defined purpose** and in this case you have to think how to **split it** into several simpler methods.

Naming Method Parameters

When naming **the parameters** of a method you can apply almost the same rules as with the methods themselves. The difference here is that it is good for the names of the parameters to use a noun or a couple of an adjective and a noun, and when naming the parameters, we use the **lowerCamelCase**

convention, i.e. **each word except for the first one starts with uppercase**. We should note that it is a good practice that the name of the parameter **shows** what **unit** is used when working with it.

A few examples for **correctly named parameters**:

- `firstName, report, speedKmH, usersList, fontSizeInPixels, font`

A few examples for **incorrectly named parameters**:

- `p, p1, p2, populate, LastName, last_name`

Good Practices When Working with Methods

Let's remind you that a method should do **only one** defined **task**. If this cannot be done, you must think how to **split** the method into a few, smaller ones. As we already said the name of the method should be clear and should describe its purpose. Another good practice in programming is to **avoid** methods, which are longer than a typical screen size (approximately). If the code still becomes large it is recommended to **split** it into several, shorter methods, as in the example below.

```
static void PrintReceipt()
{
    PrintHeader();
    PrintBody();
    PrintFooter();
}
```

Code Structure and Formatting

When writing methods, we should be careful to use correct **indentation** (moving blocks of the code to the right).

Example for **correctly** formatted C# code:

```
static void Main()
{
    // some code here...
    // some more code...
}
```

Example for **incorrectly** formatted C# code:

```
static void Main() {
    // some code...
    // some more code...
}
```

When the declaration line of the method is **too long**, it is recommended to split it into several lines, as each line after the first one is two tabulations to the right of the first one (for better readability):

```
static double GetTriangleArea(double length,
                               double height)
{
    return (length * height) / 2;
}
```

Another good practice when writing code is **to leave an empty line** between the methods, after loops and conditional statements. Also try to **abstain** from writing **long lines and complicated expressions**. With time you will see that this makes the readability better and saves time.

It is also recommended to always **use curly brackets for the bodies of conditional statements and loops**. The brackets not only improve readability, but also reduce the possibility to make a mistake and the program to run incorrectly.

Exercises: Methods

In order to learn in practice what we have learned about methods we will **solve a few problems**, in which it will be required to **write methods** with certain functionality and after that to invoke them by passing them data, read from the console.

What We Learned in This Chapter?

Before starting, let's review what we have learned about the **methods in C#**:

- We learned that the **purpose** of methods is to **split** big programs with a lot of lines of code into smaller, shorter tasks.
- We introduced ourselves with the **structure** of methods, how to **declare** them and **invoke** them by their name.
- We went over examples for methods with **parameters** and how to use them in our program.
- We learned what **signature** and **return value** of a method is and also what is the purpose of the operator **return**.
- We introduced ourselves with the **good practice** when working with methods, how to name them and their parameters, how to format code, etc.

Defining a Method

This is how we **define a method**, which takes a **parameter** and **returns** a value:

```
static double CircleArea(double radius)
{
    return Math.PI * radius * radius;
}
```

The above example defines a method called "**CircleArea**", which takes as a parameter a number (**double**) and returns as result a number (**double**).

Invoking a Method

This is how we **invoke a method**, pass a parameter value (**argument**) for the invocation and process the returned value:

```
Console.WriteLine("a = {0}, area = {1}", 5.33, CircleArea(5.33));
// a = 5.33, area = 89.2491915365671

Console.WriteLine("a = {0}, area = {1}", 9.999, CircleArea(9.999));
// a = 9.999, area = 314.0964366475
```

Problem: "Hello, Name!"

Write a method which takes a name as a parameter and prints on the console "Hello, *{name}*!".

Sample Input and Output

Input	Output
Peter	Hello, Peter!

Hints and Guidelines

Define a method **PrintName(string name)** and implement it, after which read a name from the console in the main program and invoke the method by feeding it the name.

Testing in the Judge System

Test your solution here: <https://judge.softuni.org/Contests/Practice/Index/594#7>.

Problem: Min Method - Return the Smaller Number

Create a method **GetMin(int a, int b)**, which returns the smaller of given two numbers. Write a program, which takes as input three numbers and prints the smallest of them. Use the method **GetMin(int, int)**, which you have already created.

Sample Input and Output

Input	Output	Input	Output
1		-100	
2	1	-101	
3		-102	

Hints and Guidelines

Define a method **GetMin(int a, int b)** and implement it, after which invoke it from the main program as shown below. In order to find the minimum of three numbers, first find the minimum of the first two and then the minimum of the result and the third number:

```
var min = GetMin(GetMin(num1, num2), num3);
```

Testing in the Judge System

Test your solution here: <https://judge.softuni.org/Contests/Practice/Index/594#8>.

Problem: String Repeater

Create a method **RepeatString(string str, int count)**, which takes parameters of type **string** and an integer **n** and returns the string, repeated **n** times. After this print the result on the console.

Sample Input and Output

Input	Output	Input	Output
str 2	strstr	roki 6	rokirokirokirokirokiroki

Hints and Guidelines

In the method below, inside the loop, append the input string to the result, that you will finally return:

```
static string RepeatString(string str, int count)
{
    string repeatedString = string.Empty;
```

```

for (int i = 0; i < count; i++)
{
    // TODO
}
return repeatedString;
}

```

Keep in mind that in C# concatenating strings in loops leads to bad performance and is not recommended. Learn more at: <https://docs.microsoft.com/dotnet/api/system.text.stringbuilder#the-string-and-stringbuilder-types>.

Look for more effective solutions here: <https://stackoverflow.com/questions/411752>.

Testing in the Judge System

Test your solution here: <https://judge.softuni.org/Contests/Practice/Index/594#9>.

Problem: Nth Digit

Create a method **FindNthDigit(number, index)**, which takes a number and index N as parameters and prints the Nth digit of the number (counting from right to left and starting from 1). After that print the result on the console.

Sample Input and Output

Input	Output
83746 2	4

Input	Output
93847837 6	8

Input	Output
2435 4	2

Hints and Guidelines

In order to do the algorithm use a **while** loop, until the given number equals 0. At each iteration of the **while** loop check if the current index of the digit is equal to the index you are looking for. If it is, return as a result the digit at this index (**number % 10**). If not, remove the last digit in the number (**number = number / 10**). You should count which digit you are checking by index (from right to left and starting from 1). When you find the number, return the index.

Testing in the Judge System

Test your solution here: <https://judge.softuni.org/Contests/Practice/Index/594#10>.

Problem: Integer to Base

Write a method **IntegerToBase(number, toBase)**, which takes as parameters an integer and a base of a numeral system and returns the integer converted to the given numeral system. After this the result should be printed on the console. The input number will always be in decimal numeral system, and the base parameter will be between 2 and 10.

Sample Input and Output

Input	Output
3 2	11

Input	Output
4 4	10

Input	Output
9 7	12

Hints and Guidelines

In order to solve the problem, we will declare a string, in which we will keep the result. After this we need to do the following calculations to convert the number.

- Calculate **the remainder** of the number, divided by the base.
- Insert **the remainder** in the beginning of the string.
- Divide the number to the base.
- Repeat the algorithm, until the input integer reaches 0.

Write the missing logic in the method below:

```
static string IntegerToBase(int number, int toBase) {
    string result = "";
    while (number != 0) {
        // Implement the missing conversion logic
    }
    return result;
}
```

Testing in the Judge System

Test your solution here: <https://judge.softuni.org/Contests/Practice/Index/594#11>.

Problem: Notifications

Write a program, which takes an integer **n** and **n** **input messages** and prints **n** **output messages**, based on the input. For each message read a few lines. Each message starts with **messageType**: “**success**”, “**warning**” or “**error**”:

- If **messageType** is “**success**” read **operation + message** (each from a new line).
- If **messageType** is “**warning**” read only **message** (from a new line).
- If **messageType** is “**error**” read **operation + message + errorCode** (each from a new line).

Print on the console **each read message**, formatted depending on its **messageType**. After the headline of the message print as much “=”, as the length of the printed headline and print an **empty line** after each message (to understand in detail look at the examples).

Sample Input and Output

Input	Output
4	Error: Failed to execute credit card purchase. =====
error	Reason: Invalid customer address. Error code: 500.
credit card purchase	
Invalid customer address	
500	Warning: Email not confirmed. =====
warning	
Email not confirmed	
success	Successfully executed user registration. =====
user registration	User registered successfully.
User registered successfully	

Input	Output
warning Customer has not email assigned	Warning: Customer has not email assigned. =====

The problem should be solved by defining the following four methods: **ShowSuccessMessage()**, **ShowWarningMessage()**, **ShowErrorMessage()** and **ReadAndProcessMessage()**, so that only the last method is invoked by the **Main()** method:

```
static void ShowSuccessMessage(string operation, string message) { ... }
static void ShowWarningMessage(string message) { ... }
static void ShowErrorMessage(string operation, string message, int errorCode) { ... }
static void ReadAndProcessMessage() { ... }
```

Hints and Guidelines

Define and implement the four shown methods. In **ReadAndProcessMessage()** read the type of message from the console and according the read type read the rest of the data (one, two or three more lines). After that invoke the method for printing the given type of message.

Testing in the Judge System

Test your solution here: <https://judge.softuni.org/Contests/Practice/Index/594#12>.

Problem: Numbers to Words

Write a method **Letterize(number)**, which reads an integer and prints it in words in English according to the conditions below:

- Print in words the hundreds, the tens and the ones (and the eventual minus) according to the rules of the English language.
- If the number is larger than **999**, you must print "too large".
- If the number is smaller than **-999**, you must print "too small".
- If the number is **negative**, you must print "minus" before it.
- If the number is not built up of three digits, you shouldn't print it.

Sample Input and Output

Input	Output
3 999 -420 1020	nine-hundred and ninety nine minus four-hundred and twenty too large
4 311 418 509 -9945	three-hundred and eleven four-hundred and eighteen five-hundred and nine too small

Input	Output
2 15 350	fifteen three-hundred and fifty
3 500 123 9	five-hundred one-hundred and twenty three nine

Hints and Guidelines

We can first print **the hundreds** as a text – (the number / 100) % 10, after that **the tens** – (the number / 10) % 10 and at the end **the ones** – (the number % 10).

The first special case is when the number is exactly **rounded to 100** (e.g. 100, 200, 300 etc.). In this case we print "one-hundred", "two-hundred", "three-hundred" etc.

The second special case is when the number formed by the last two digits of the input number is **less than 10** (e.g. 101, 305, 609 etc.). In this case we print "one-hundred and one", "three-hundred and five", "six-hundred and nine" etc.

The third special case is when the number formed by the last two digits of the input number is **larger than 10 and smaller than 20** (e.g. 111, 814, 919 etc.). In this case we print "one-hundred and eleven", "eight-hundred and fourteen", "nine-hundred and nineteen" etc.

Testing in the Judge System

Test your solution here: <https://judge.softuni.org/Contests/Practice/Index/594#13>.

Problem: String Encryption

Write a method **Encrypt(char letter)**, which encrypts a given letter in the following way:

- It takes the first and the last digit from the ASCII code of the letter and concatenates them into a string, which will represent the result.
- In the beginning of the string, which represents the result, we will insert the symbol which matches the following condition:
 - ASCII code of the letter + the last digit of the ASCII code of the letter.
- After that in the end of the string, which represents the result, you concatenate the character which matches the following condition:
 - ASCII code of the letter - the first digit of the ASCII code of the letter.
- The method should return the encrypted string.

Example:

- $j \rightarrow p16i$
 - ASCII code of **j** is **106** → First digit – **1**, last digit – **6**.
 - We concatenate the first and the last digit → **16**.
 - At **the beginning** of the string, which represents the result, concatenate the symbol, which you get from the sum of the ASCII code + the last digit → $106 + 6 \rightarrow 112 \rightarrow p$.
 - At **the end** of the string, which represents the result, concatenate the symbol, which you get from subtracting the ASCII code – the first digit → $106 - 1 \rightarrow 105 \rightarrow i$.

Using the method shown above, write a program which takes **a sequence of characters**, **encrypts them** and prints the result on one line.

The input data will always be valid. The Main method must read the data given by the user – an integer **n**, followed by a character for each of the following **n** lines.

Encrypt the symbols and add them to the encrypted string. In the end, as a result, you must print **an encrypted string** as in the following example.

Example:

- S, o, f, t, U, n, i → V83Kp11nh12ez16sZ85Mn10mn15h

Sample Input and Output

Input	Output	Input	Output
7 S o f t U n i	V83Kp11nh12ez16sZ85Mn10mn15h	7 B i r a H a x	H66<15hv14qh97XJ72Ah97x x10w

Hints and Guidelines

Firstly, we will give a value of `string.Empty` to the `string`, which will keep the result. We must recur a loop `n` times, so that in each iteration we will add the encrypted symbol to the result string.

In order to find the first and the last digit of the ASCII code, we will use the same algorithm that we used to solve "Integer to Base".

Testing in the Judge System

Test your solution here: <https://judge.softuni.org/Contests/Practice/Index/594#14>.

High-quality education profession and job for software engineers

- ✓ Free start for **absolute beginners**
- ✓ Choose among the **most in-demand** professions in the software industry
- ✓ Assistance for **career start**
- ✓ **Top trainers**, mentors and active learning **community**



SoftUni provides practical, modern and **innovative education** for programming, IT, design, marketing and digital skills for thousands of young people. The "**Software University**" program builds true **software engineering professionals**.

Join the end-to-end **software engineering learning program** at SoftUni to master the most in-demand programming concepts, languages, software technologies and skills through a modern teaching methodology, **interactive learning platform** and tons of practical exercises and projects. Our **curriculum** is developed together with the **IT companies** to closely match the industry demands.

SoftUni works directly with the **companies** from the software industry to **find a job** for its students and to make them successful software engineers.

The path of the student at SoftUni



Apply now

softuni.org/apply

Chapter 11. Tricks and Hacks

In the current chapter we are going to see some tricks, hacks and techniques, which will make our work with C# easier in the Visual Studio IDE. In particular, we will see:

- How to properly **format our code**
- Conventions for **naming elements in the code**
- Some **keyboard shortcuts**
- Some **code snippets**
- Techniques to **debug our code**

Code Formatting

The right formatting of our code will make it **easier to read and understand** in case someone else needs to work with it. This is important, because in practice we will need to work in a team with other people and it is highly important to write our code in a way that our colleagues can **quickly understand**.

There are some defined rules for correct formatting of the code, which are collected in one place and are called **conventions**. The conventions are a group of rules, generally accepted by the programmers using a given language, which are massively used. These conventions help building norms in given languages – what is the best way to write and what are good practices. It is accepted that if a programmer follows them then his code is easy to read and understand.

The C# language is made by **Microsoft** and they are the people who define the best practices for writing. You should know that even if you don't follow the conventions given by Microsoft, your code will work (as long as it is properly written), but it will not be easy to understand. This, of course, is not fatal at base level, but the faster you get used to writing quality code the better.

The Official C# Code Conventions

The official **C# code conventions** by Microsoft are published in the "C# Coding Conventions" article in the .NET documentation and in this book we shall follow them: <https://docs.microsoft.com/dotnet/csharp/programming-guide/inside-a-program/coding-conventions>.

For code formatting Microsoft recommends **curly brackets {}** to be on a separate line and just below the construction to which they apply, as in the example below:

```
if (someCondition)
{
    Console.WriteLine("Inside the if statement");
}
```

You can see that the command **Console.WriteLine(...)** in the example is **offset by 4 white spaces (one tab)**, which is also recommended by **Microsoft**. If given construction with curly brackets is offset by one tab, then **the curly brackets {} must be in the beginning of the construction**, as in the example below:

```
if (someCondition)
{
    if (anotherCondition)
    {
        Console.WriteLine("Inside the if statement");
    }
}
```

Below you can see an example for **badly formatted code** according to the accepted conventions for writing code in C#:

```
if(someCondition){
    Console.WriteLine("Inside the if statement");}
```

The first thing that we see is **the curly brackets {}**. The first (opening) bracket should be just below **the if condition**, and the second (closing) bracket – **below the command Console.WriteLine(...)**, at a new and **empty line**. In addition, the command inside the **if** construction should be **offset by 4 white spaces (one tab)**. Just after the keyword **if** and before the condition you should put **a space**.

The same rule applies for **the for loops and all other constructions with curly brackets {}**. Here are some more examples:

Correct formatting:

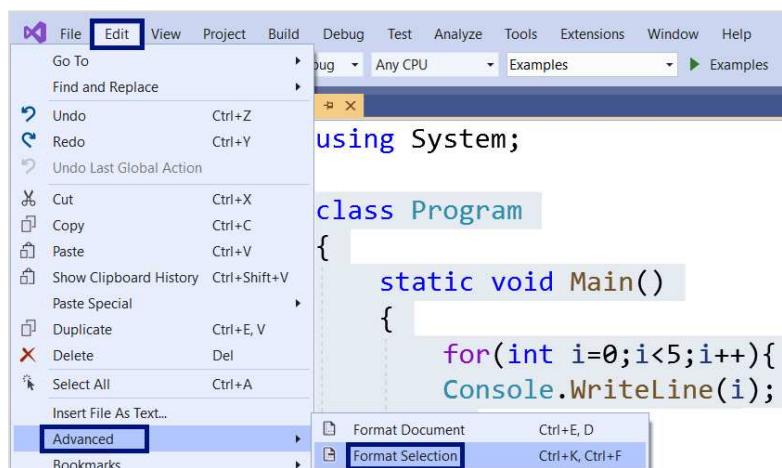
```
for (int i = 0; i < 5; i++)
{
    Console.WriteLine(i);
}
```

Wrong formatting:

```
for(int i=0;i<5;i++){
    Console.WriteLine(i);
}
```

Code Formatting Shortcuts in Visual Studio

For your comfort there are **keyboard shortcuts** in **Visual Studio**, which we will explain later in this chapter, but for now we are interested in two specific combinations. The first is for formatting **the code in the entire program**, and the second one – for formatting **a part of the code**. If we want to format **the entire code**, we need to press **[CTRL + K + D]**. In case we need to format only **a part of the code**, we need to mark **this part** with the mouse and press **[CTRL + K + F]**.



Let's use **the wrongly formatted example** from earlier:

```
for(int i=0;i<5;i++){
    Console.WriteLine(i);
}
```

If we press **[CTRL + K + D]**, which is the combination to **format the whole document**, we will have a code, formatted according to **the accepted conventions for C#**, which will look as follows:

```
for (int i = 0; i < 5; i++)
{
    Console.WriteLine(i);
}
```

This key combination in Visual Studio can help us if we work with a badly formatted code.

Naming Code Elements

In this section we will focus on the accepted conventions for naming projects, files and variables, defined by Microsoft.

Naming Projects and Files

It is recommended to use a descriptive name **for naming projects and files**, which suggests **the role** of the respective file / project and at the same time the **PascalCase convention** is also recommended. This is a **convention for naming** elements, in which each word, including the first one, starts with **an uppercase character**, for example **ExpressionCalculator**.

Example: this course starts with a **First steps in coding** lecture, therefore an exemplary name for the solution for this lecture can be **FirstStepsInCoding**. The same convention applies for the files in a project. If we take for example the first problem in the **First steps in coding** lecture, it is called **HelloWorld**, therefore our file in the project will be called **HelloWorld**.

Naming Variables

In programming variables keep data, and for the code to be more understandable, the name of a variable should **suggest its purpose**. Here are some recommendations for naming variables:

- The name should be **short and descriptive** and to explain what the variable serves for.
- The name should only contain the letters **a-z, A-Z, the numbers 0-9, and the symbol '_'**.
- It is accepted in C# for the variables to always **begin with a lowercase letter** and to **contain lowercase letters**, and **each next word** in them should **start with an uppercase letter** (this naming is also known as **camelCase** convention).
- You should be careful about uppercase and lowercase letters, because C# distinguishes them. For example, **age** and **Age** are different variables.
- The names of the variables cannot coincide with keywords in the C# language, for example **int** is an invalid name for a variable.



Although using the symbol `_` in the names of variables is allowed, in C# it is not recommended and is considered a bad style of naming.

Naming – Examples

Here are some examples for **well named** variables:

- `firstName`
- `age`
- `startIndex`
- `lastNegativeNumberIndex`

Here are some examples for **badly named variables**, even though the names are correct according to the C# compiler:

- `_firstName` (starts with '_')
- `last_name` (contains '_')
- `AGE` (written in uppercase)
- `Start_Index` (starts with an uppercase letter and contains '_')
- `lastNegativeNumber_Index` (contains '_')

At a first look all these rules can seem meaningless and unnecessary, but with time passed and experience gaining you will see the need for **conventions for writing quality code** in order to be able to work more easily and faster in a team. You will understand that the work with a code, which is written without complying with any rules for code quality, is annoying.

Shortcuts in Visual Studio

In the previous section we mentioned two of the combinations that are used for formatting code. One of them [**CTRL + K + D**] is used for **formatting the whole code in a file**, and the second one [**CTRL + K + F**] serves if we want to **format just a piece of the code**. These combinations are called **shortcuts** and now we will give more thorough information about them.

Shortcuts are **combinations** that give us the possibility to do some things in an **easier and faster** way, and each IDE has its shortcuts, even though most of them are recurring. Now we will look at some of the **shortcuts in Visual Studio**:

Combination	Action
[CTRL + F]	Opens the search window , by which we can search in the code .
[CTRL + K + C]	Comments part of the code.
[CTRL + K + U]	Uncomments a code , which is already commented.
[CTRL + Z]	Brings back one change (so-called Undo).
[CTRL + Y]	The combination is opposite of [CTRL + Z] (the so-called Redo).
[CTRL + K + D]	Formats the code according the default conventions.
[CTRL + Backspace]	Deletes the word to the left of the cursor.
[CTRL + Del]	Deletes the word to the right of the cursor.
[CTRL + Shift + S]	Saves all files in the project.
[CTRL + S]	Saves the current file.

More information about the **keyboard shortcuts in Visual Studio** can be found at this Web site: <https://shortcutworld.com/en/Visual-Studio/2015/win/all>.

Code Snippets in Visual Studio

In Visual Studio there are the so-called **code snippets**, which write a block of code by using a code template. For example, by writing the short code "**cw**" and then pressing [**Tab**] + [**Tab**] the

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

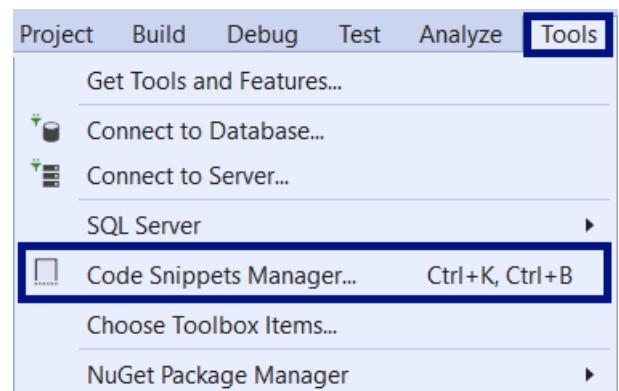
code is generated in the body of our program, in the place of the short code. This is called “unfolding a code snippet”. The “**for**” + [Tab] + [Tab] snippet works in the same way. On the figure below you can see the “**cw**” snippet in action:



Creating Your Own Code Snippet

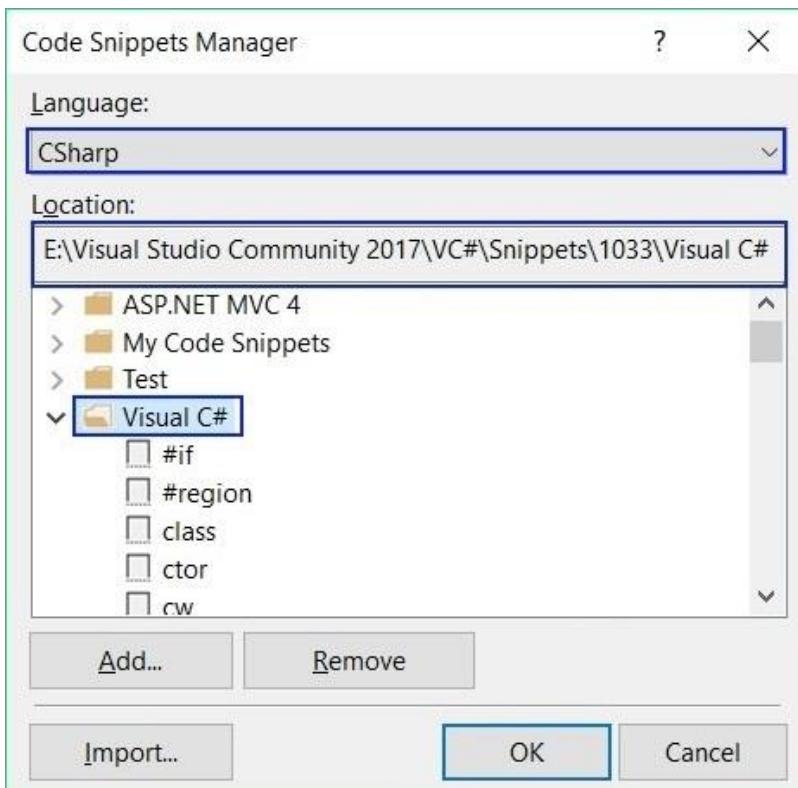
In this section we are going to show you how to **make your own code snippet**. We will see how to make a code snippet for quick typing of `Console.ReadLine()`.

In order to begin we must create a new empty project in Visual Studio and go to [Tools -> Code Snippets Manager], as shown on the screenshot on the right.



Exploring the Existing Code Snippets for C#

In the window that VS will open, we should choose **Language → CSharp**, and from the section **Locations → Visual C#**. This is where all the existing snippets for **C#** are located:



We choose a snippet, for example `cw`, we take the path to its file and open it in Visual Studio:

```

1  <?xml version="1.0" encoding="utf-8" ?>
2  <CodeSnippets
3      xmlns="http://schemas.microsoft.com/VisualStudio/2005/CodeSnippet"
4      Format="1.0.0">
5      <Header>
6          <Title>cw</Title>
7          <Shortcut>cw</Shortcut>
8          <Description>Code snippet for Console.WriteLine</Description>
9          <Author>Microsoft Corporation</Author>
10         <SnippetTypes>
11             <SnippetType>Expansion</SnippetType>
12         </SnippetTypes>
13     </Header>
14     <Snippet>
15         <Declarations>
16             <Literal Editable="false">
17                 <ID>SystemConsole</ID>
18                 <Function>SimpleTypeName(global::System.Console)</Function>
19             </Literal>
20         </Declarations>
21         <Code Language="csharp"><![CDATA[$SystemConsole$.WriteLine($end$);]]>
22     </Code>
23 </Snippet>
24 </CodeSnippet>
25 </CodeSnippets>

```

We see many things we haven't seen yet, but don't worry, we will become acquainted with them later.

Changing an Existing Snippet

To create a **new code snipped**, we shall take an existing snipped, modify its code and save it in a new snipped file.

We have to focus on the part `<Title></Title>`, `<Shortcut></Shortcut>` and the code between `CDATA[]`.

- Firstly, we will change the title in `<Title></Title>` and in the place of `cw` we will write `cr`, as this will be **the title of our snippet**.
- After that, in the section `<Shortcut></Shortcut>`, we will change what we have to write to **call our snippet** (the shortcut) from `cw` to `cr`.
- Finally, we need to change the code in the `CDATA[]` section from `WriteLine` to `ReadLine`: `CDATA[$SystemConsole$.ReadLine(end);]`.
- If you want to import some class, you may change the `<Declarations></Declarations>` section.
- If you wish, you can change accordingly also the sections `<Description></Description>` and `<Author></Author>`.

The changed file, after all described modifications, should look like this:

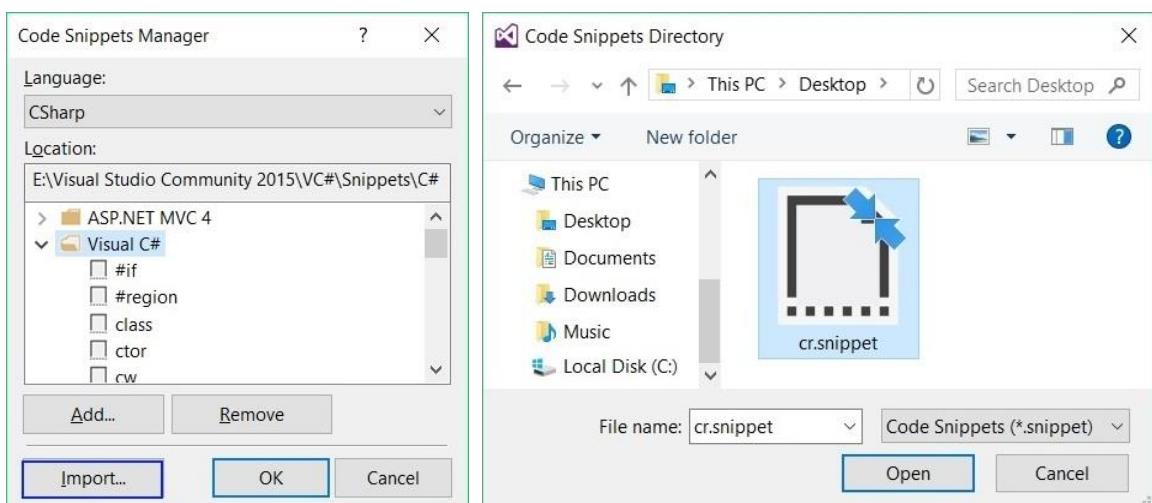
```

1  <?xml version="1.0" encoding="utf-8" ?>
2  <CodeSnippets
3      xmlns="http://schemas.microsoft.com/VisualStudio/2005/CodeSnippet">
4      <CodeSnippet Format="1.0.0">
5          <Header>
6              <Title>cr</Title>
7              <Shortcut>cr</Shortcut>
8              <Description>Code snippet for Console.ReadLine</Description>
9              <Author>Microsoft Corporation</Author>
10             <SnippetTypes>
11                 <SnippetType>Expansion</SnippetType>
12             </SnippetTypes>
13         </Header>
14         <Snippet>
15             <Declarations>
16                 <Literal Editable="false">
17                     <ID>SystemConsole</ID>
18                     <Function>SimpleTypeName(global::System.Console)</Function>
19                 </Literal>
20             </Declarations>
21             <Code Language="csharp"><![CDATA[$SystemConsole$.ReadLine($end$);]>
22             </Code>
23         </Snippet>
24     </CodeSnippet>
25 </CodeSnippets>

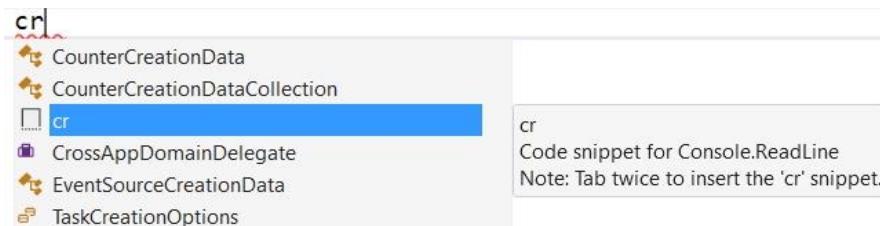
```

Saving and Testing the Code Snippet

After we have written our snippet, we should save the file in format `snippetName.snippet` (in this case `cr.snippet`) and add it to Visual Studio. Go to [Tools] -> [Code Snippet Manager] -> [Import] and choose the `cr.snippet` file that we have created:



Now when we write “`cr`” and press `[Tab]` twice in Visual Studio, our new snippet will appear, as it is shown in the screenshot below:



Code Debugging Techniques

Debugging plays an important role in the process of creating software, which is to allow us to **follow the implementation** of our program **step by step**. With this technique we can **follow the values of the local variables**, because they are changing during the execution of the program, and to **remove possible errors** (bugs). The process of debugging includes:

- **Finding** the problems (bugs).
- **Locating** the code, which causes the problems.
- **Correcting** the code, which causes the problems, so that the program works correctly.
- **Testing** to make sure that the program works correctly after the corrections we have made.

Debugging in Visual Studio

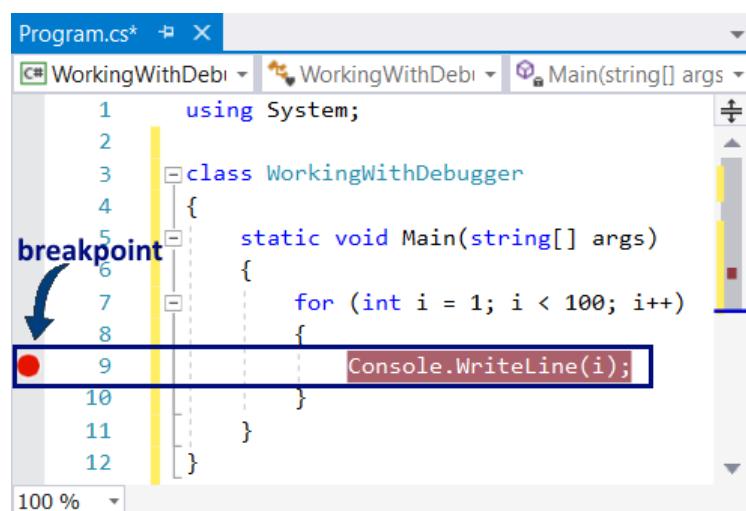
Visual Studio gives us a **built-in debugger**, thanks to which we can place **breakpoints** at places we have chosen. When it reaches a **breakpoint**, the program stops running and allows **step-by-step running** of the remaining lines. Debugging allows us to **get in the details of the program** and see where exactly the errors occur and what is the reason for this.

In order to demonstrate how to use the debugger in VS, we will use the following C# program:

```
static void Main(string[] args)
{
    for (int i = 0; i < 100; i++)
    {
        Console.WriteLine(i);
    }
}
```

We will place a **breakpoint** (a stopper) at start of the line holding **Console.WriteLine(...)**. For this we will need to move the mouse cursor to the line, which prints on the console, and press the [F9] key. A red **breakpoint** will appear, pointing where the program will **stop** running (see the screenshot).

Now the program is **configured for debugging** in the Visual Studio IDE, using breakpoints. It is time to start it through the debugger (in the so called “debug mode”) and trace its execution step by step.



Starting the Program in Debug Mode

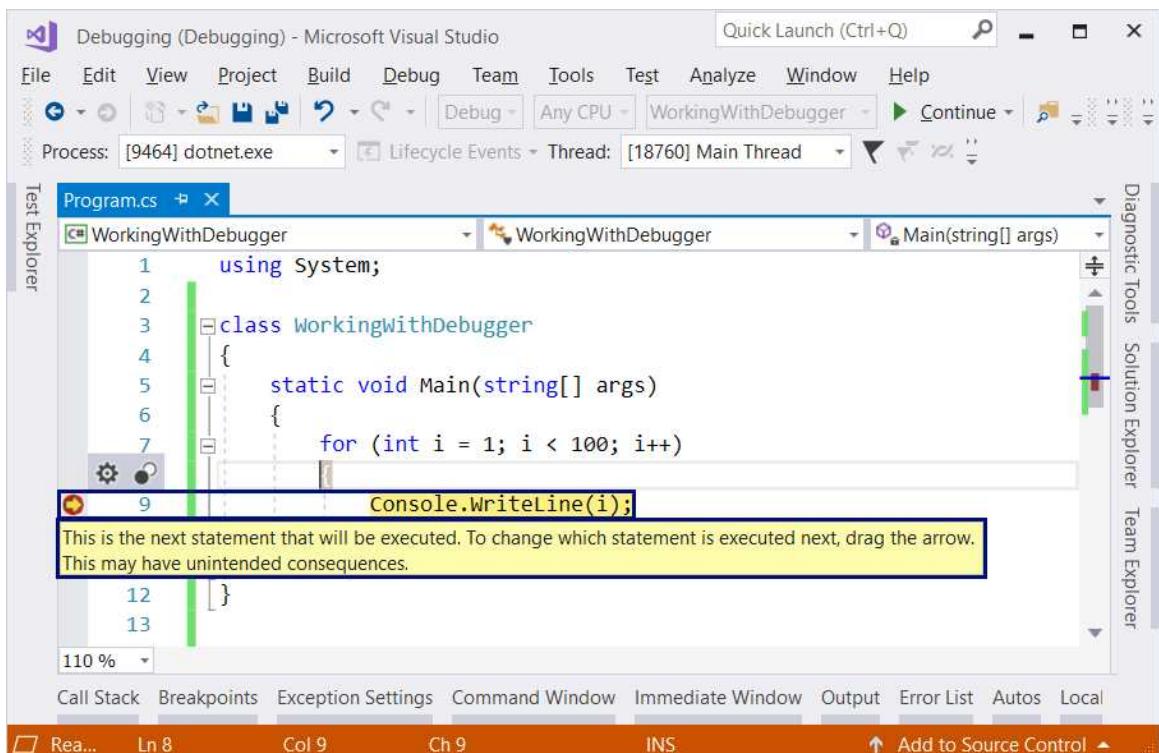
In order to start the program in debug mode, we choose [Debug] -> [Start Debugging] or press the [F5] key (see the screenshot).

After starting the program, we can see that it will **stop executing** at line 11, where we placed our breakpoint.

The code in the current line when the debugger is stopped **is colored in yellow** and we can **run it step by step**. In order to execute **the next line**, we use the [F10] or the [F11] key.

We can see that **the code on the current line hasn't executed yet** and it is displayed in yellow. It will execute when we go ahead with the debugging the next line.

The screenshot below shows the integrated **Visual Studio debugger**, stopped at the breakpoint and waiting for the developer to decide what to do (e.g. execute the next command or stop the program):



From the **Locals** window we can observe **the local variables** over the time or **modify their values** at runtime. In order to open the window, you must choose [Debug] -> [Windows] -> [Locals].

Name	Value	Type
args	{string[0]}	string[]
i	4	int

Tricks for C# Developers

In this section we will recall some **tricks and techniques** in programming with C#, already seen in this book, which can be very useful if you attend an exam for beginner programming.

Inserting Variable Values in Strings

In programming we often need to **combine text with variable values** to obtain a string value, e.g.

```
var text = "some text";
Console.WriteLine("{0}", text);
// This will print on the console "some text"
```

In this case we are using a **placeholder** – **{x}**, where **x** is a number (larger than or equal to 0), corresponding to the position on which we have placed our variable. Therefore, if we insert two variables, we will have one placeholder, which will be **{0}** and it will keep the value of **the first variable** and another one – **{1}**, which will keep the value of **the second variable**. For example:

```
var text = "some text";
var number = 5;
Console.WriteLine("{0} {1} {0}", text, number);
// This will print "some text 5 some text"
```

In this example we can see that we can insert **not only text variables**. We can also use a given variable **several times** and for this we put the number which **corresponds with the position of the variable** in the placeholder. In this case on position zero is the variable **text**, and at first position is the variable **number**. At the beginning, numbering can be confusing, but you need to remember that in programming counting starts from 0.

Formatting with 2 Digits After the Decimal Point

When we print numbers, we often need to round them to 2 digits after the decimal point, e.g.

```
var number = 5.432432;
Console.WriteLine(Math.Round(number, 2));
// This will print on the console "5.43"
```

Rounding Numbers

To round numbers, we may use the **Math.Round(...)** method, which takes 2 parameters:

- the first one is **the number we want to round**
- the second one is the number that determines **how many digits after the decimal point we want to round to** (this should always be an integer)

If we want to round the number to **2 digits after the decimal point** and the third digit is lower than 5, as in the example above, the rounding is down, but if the third digit is equal or bigger than 5 – the rounding is up as in the example below:

```
var number = 5.439;
Console.WriteLine(Math.Round(number, 2));
// This will print on the console "5.44"
```

Other Rounding Methods

In case we always want to round down instead of `Math.Round(...)` we can use another method `Math.Floor(...)`, which always rounds down, but also always rounds to an integer. For example, if we have the number 5.99 and we use `Math.Floor(5.99)`, we will get the number 5.

We can also do the exact opposite – to always round up using the method `Math.Ceiling(...)`. Again, if we have for example 5.11 and we use `Math.Ceiling(5.11)`, we will get 6. Here are some examples:

```
var numberToFloor = 5.99;
Console.WriteLine(Math.Floor(numberToFloor));
// This will print on the console 5

var numberToCeiling = 5.11;
Console.WriteLine(Math.Ceiling(numberToCiling));
// This will print on the console 6
```

Rounding with a Placeholder

```
var num = 5.432424;
Console.WriteLine("{0:f2}", num);
```

In this case after the number we add `:f2`, which will limit the number to 2 digits after the decimal point and will work like `Math.Round(...)`. You should keep in mind that the number after the letter `f` means to how many digits after the decimal point the number is rounded (i.e. it can be `f3` or `f5`).

How to Write a Conditional Statement?

The conditional `if` construction contains the following elements:

- Keyword `if` + a Boolean expression (condition)
- Body of the conditional construction
- Optional: `else` clause

Example:

```
if (condition)
{
    // body
}
else (condition)
{
    // body
}
```

To make it easier we can use a code snippet for an `if` construction:

- `if` + [Tab] + [Tab]

How to Write a 'For' Loop?

For a `for` loop we need a couple of things:

- **Initializing block**, in which the counter variable is declared (**var i**) and its initial value is set
- **Condition** for repetition (**i <= 10**)
- Loop variable (counter) **updating statement** (**i++**)
- **Body** of the loop, holding statements

Example:

```
for (var i = 0; i < 5; i++)  
{  
    // body  
}
```

To make it easier we can use a code snippet for a **for** loop:

- **for + [Tab] + [Tab]**

What We Learned in This Chapter?

In the **current** chapter we learned how to **correctly format** and name the elements of our **code**, some **shortcuts** in Visual Studio, some **code snippets**, and we analyzed how to **debug** the code.

Conclusion

If you have **read the entire** book and you've solved all the problems from the exercises and reached the present conclusion, **congratulations!** You've already made the **first step** in learning the **profession of a programmer**, but there is a **long way** to go until you become **really good** and make **software writing** your **profession**.

Developer Skills

Remember the [**four main groups of skills**](#) (see the Preface chapter) that each programmer must have in order to work in the industry:

- Skill #1 – **writing the program code** (20% of programmer's skills) – covered to a large degree by this book, but you must learn additional basic data structures, classes, functions, strings and other elements of code writing.
- Skill #2 – **algorithmic thinking** (30% of programmer's skills) – covered partially by this book and developed mostly by solving a large amount of diverse algorithmic problems.
- Skill #3 – **fundamental understanding of the profession** (25% of programmer's skills) – acquired for a few years in combination with learning and practice (reading books, watching video lessons, attending courses and mostly by writing diverse projects in various technological areas).
- Skill #4 – **programming languages and software technologies** (25% of programmer's skills) – acquired in a long period of time, by a lot of practice, consistent reading and writing projects. Such knowledge and skills quickly get outdated and need to be updated frequently. Good programmers are involved in studying new technologies every day.

This Book is Only the First Step!

The **present** book on programming basics is just the **first step** in building the skills of a programmer. If you were able to solve **all problems**, this means you have **obtained valuable knowledge** in the programming principles with C# on a **basic level**. You are about to start **in-depth** studying of programming, and develop **your algorithmic thinking**, and then add **technological knowledge** regarding the C# language and the .NET ecosystem (.NET Framework, .NET Core, Entity Framework, ASP.NET, etc.), front-end technologies (HTML, CSS, JavaScript) and many other concepts, technologies and instruments for software development.

If you were **not able** to solve all problems or a large part of them, go back and solve them! Remember that **becoming a programmer** requires **a lot of work and efforts**. This profession is not for lazy people. There is no way to learn it, unless **you seriously practice** programming for years!

As we already explained, the first and basic skill of a programmer is **to learn to write code** with ease and pleasure. This is namely the mission of this book: to teach you how to code. We recommend you, besides reading the book, to enroll in the [**practical course "Programming Basics" at SoftUni**](#) (<https://softuni.org>), which is offered for free, in on-site or online format of training.

How to Proceed After This Book?

This book **gives you solid grounds**, thanks to which it will be easy for you to continue developing as programmers. If you wonder how to continue your development, you have the following possibilities:

- to study for a **software engineer** at **SoftUni** and make programming your profession;
- to continue developing as a programmer **in your own way**, for example through self-training or via online lessons;
- to **stay at coder level**, without going more seriously into programming.

Study Software Engineering in SoftUni

The first, and respectively recommended option to master fully and on high level the profession of a "software engineer", is to start your training via the **end-to-end SoftUni program for software engineers**: <https://softuni.org>. The SoftUni curriculum and the interactive learning platform for developers are carefully developed by **Dr. Svetlin Nakov and his team**, in order to provide you consequently and with gradually increasing complexity all the skills that a software engineer must have, in order to **start a career as a software developer** in an IT company.

```

class FreezingWeather
{
    static void Main()
    {
        string color = "red";
        if (color == "red")
            Console.WriteLine("tomato");
        else
            Console.WriteLine("banana");
        Console.WriteLine("lemon");
    }
}

```

Training Duration in SoftUni

The training in SoftUni has a duration of **1-2 years** (depending on the profession and the selected specializations) and during that period it is normal to reach a good starting level (junior developer), but this is **only if you study seriously** and write code intensely every day. Upon having good grades, a typical student **starts a job around the middle of the training (after around 1.5 years)**. Thanks to the well-developed partners network, **the career center of SoftUni offers work** in a software or IT company to all SoftUni students who have very good or excellent grades. **Starting a job** in the major in case of having good grades at SoftUni, combined with willingness to work and reasonable expectations towards the employers, is almost guaranteed.

Becoming a Programmer Takes at Least a Year!

Keep in mind that **to become a programmer takes a lot of efforts**, writing tens of thousands of lines of code, and solving hundreds, even thousands of practical problems, and this takes years! If someone offers you "**an easier program**" and promises you to become a programmer and start working within 3-4 months, then either they are **lying** to you, or they will give you such a low level, that **companies won't even take you as a trainee**, even if you pay to the company that is wasting its time with you. There are exceptions, of course, for example if you are not starting from scratch, or if you have extremely well-developed engineering thinking, or if you apply for a very low position (for example

technical support), but in general, you cannot become a programmer if you haven't spent at least 1 year of intense learning and code writing!

The Entrance Exam in SoftUni

In order to enroll at SoftUni you need to attend an **entrance exam** in "Programming Basics" on the material from this book. If you easily solve the problems in this book, then you are ready for the exam. Also, pay attention to the chapters on **preparation for the practical exam in programming**. They will give you a good idea of the level of difficulty of the exam and the types of tasks that you need to learn solving.

If the tasks from the book and the preparation examples are hard for you, then you **need more preparation**. Enroll for the [free course in "Programming Basics"](#) or go through the book carefully one more time, without skipping solving **the problems in any of the studied topics!** You must learn how to **solve them with ease**, without helping yourselves with the guidelines and the sample solutions.

The SoftUni Curriculum for Software Engineers

What follows after the entrance exam is a **serious curriculum** in the SoftUni program for training software engineers. It is formed as a sequence of **modules in a number of courses** in programming and software technologies, fully directed towards gaining fundamental knowledge in software development and acquiring **practical skills for working** as a programmer with the most contemporary software technologies. Students are given a choice between **a number of professions** and specializations focused on C#, Java, JavaScript, Python, PHP and other languages and technologies. Each profession is trained in a number of modules with 4 months duration, and each module consists of 2 or 3 courses. The classes are divided into **theoretical preparation** (30%) and **practical exercises, projects and trainings** (70%), and each course ends with a practical exam or practical academic project.

How Many Hours per Day Does the Training Take?

The training for software engineers at SoftUni is a **very serious occupation** and you need to spend on it **at least 4-5 hours every day**, preferably your entire attention and time. Combining **working and training** is not always successful, but if you work something easy and you have a lot of spare time, it is a good option. SoftUni is an appropriate option for **school students, university students and people who work**, but it is best if you assign your entire time to your training and mastering the profession. It will not work if you spend 2 or 4 hours a week on it!

The forms of training at SoftUni are **on-site** (the better choice) and **online** (if you don't have another option). In both forms of training, in order to learn the program in the curriculum (that is required by software companies for starting a job), you need **a lot of learning**. You just need to **find the time for it!** Reason #1 for having hard time on the road to the profession in SoftUni is not spending enough time for the training: as a minimum you need to spend at least 20-30 hours a week.

SoftUni for People Who Work and Study

We recommend to everyone who gets **excellent score at the SoftUni entrance exam** and are really passionate about making programming their profession, to leave the rest of their commitments aside and **spend their entire time** on learning the profession of a "software engineer" and start making a living through it.

- For people who **work** this means quitting their job (and getting a loan or decreasing their expenses, in order to spend with a lower income a period of 1-2 years until they start working in the new profession).

- For people who **study** in a traditional university, this means to move significantly their focus towards programming and the practical courses in SoftUni, by decreasing the time spent in the traditional university.
- For **unemployed** people this is an excellent chance to assign their entire time, power and energy on acquiring a new, perspective, well paid and highly sought profession, that will give them good life quality and a long-term prosperity.
- For **students** in secondary schools and high schools this means **giving a priority** to what is more important in their development: studying practical programming in SoftUni, that will give them a profession and a job, or giving their full attention to the traditional education system or combining smartly both undertakings. Unfortunately, often priorities are determined by parents, and we don't have a solution for these cases.

We recommend to all who **cannot get an excellent score at the SoftUni entrance exam** to spend more time on better learning, understanding, and most of all, practicing the material studied in the present book. If you cannot easily solve the problems in this book, you will not be able to cope with programming and software development in the future.

Do not skip the programming basics! Do not under any circumstances make bold decisions and quit your job or the traditional university, making great plans for your future profession of a software engineer, if you don't have an excellent grade at the SoftUni entrance exam! It measures if programming is suitable for you, to what extend you like it and if you are motivated to study it seriously, and work this for years every day with joy and pleasure.

Study Software Engineering in Your Own Way

Another possibility to develop after this book is to **continue studying programming outside of SoftUni**. You can enroll or subscribe to **video trainings** that go into more details in programming with C# or other languages and development platforms. You can **read books** on programming and software technologies, follow **online tutorials** and other online resources – there are plenty of free materials on the Internet. However, keep in mind that the most important thing towards the profession of a programmer is **to do practical projects!**

You **cannot become a programmer without a lot of code writing and intense practicing**. Allocate **sufficient time** to it. You cannot become a programmer for a month or two. On the Internet you will find a wide variety of **free resources**, such as books, manuals, video lessons, online and on-site courses on programming and software development. However, you need to invest **at least a year or two** to acquire a foundation level, needed for starting a job.

After you gain some experience, find a way to start **an internship in a company** (which will be almost impossible unless you'd spent at least a year of intense code writing before that) or come up with **your own practical project**, on which you need to spend a few months, even a year, in order to learn based on the trial-and-error principle.



Keep in mind that there are many ways to become a programmer, but they all have something in common: **intense code writing and years of practice!**

Recommended Resources for Developers

A **huge amount of resources** is available on the Web for developers: online trainings, courses, tutorials, books, interactive training sites, etc.

We cannot mention all of them, because they change over the time and because the list might be huge. What we recommend is to **join the developer communities** in your region, because this will help you a lot when you study.

Online Communities for Beginners in Programming

Regardless of the path you have chosen, if you are seriously involved in programming, we recommend subscribing to specialized **online forums, discussion groups and communities**, from which you can get assistance by your colleagues and track the novelties in the software industry.

If you will study programming seriously, **surround yourselves with people who are involved in programming** seriously. Join **communities of software developers**, attend software conferences, go to events for programmers, find friends with whom you can talk about programming and discuss problems and bugs, find environment that can help you. In most large towns there are free events for programmers, a few times a week. In smaller localities you have the Internet and an access to the entire online community.

Here are some recommended **resources** that will be useful for your development as a programmer:

- <https://softuni.org> – official **website of SoftUni**. In it you will find free (and not only) courses, seminars, video tutorials and trainings in programming, software technologies and digital competences.
- <https://fb.com/softuni.org> – official **Facebook page of SoftUni**. By it you will learn regarding new courses, seminars and events related to programming and software development.
- <https://introprogramming.info> – official website of the **books "Programming Basics"** with **C#** and **Java** by Dr. Svetlin Nakov and his team. The books examine in-depth programming basics, basic data structures and algorithms, object-oriented programming, and other basic skills, and are an excellent continuation for reading after the current book. However, **besides reading, you need to do intense code writing**, do not forget that!
- <https://stackoverflow.com> – **Stack Overflow** is one of the **widest** discussion forums for programmers worldwide, where you will find assistance for any possible question in the world of programming. If you are fluent in English, look up at StackOverflow and ask your questions there.
- <https://udemy.com> – **Udemy** is one of the biggest **marketplaces** for technical trainings, offered free or at affordable prices.
- <https://meetup.com/find/tech> – look for **tech meetups** around your city and involve in communities that you like. Most of the tech meetups are free and newbies are welcome.

Good Luck to All!

On behalf of the entire authors' team, we **wish you endless success in the profession and in your life!** We will be really happy if we have helped you get **passionate about programming** and we have inspired you to go bravely towards becoming a "software engineer", which will bring you a good job that you will work with pleasure, give you a quality life and prosperity, as well as amazing perspectives for development and possibilities for making impressive projects with inspiration and passion.

Sofia, September 1, 2019



In your hands you hold something more than a **programming book**, textbook or tutorial. This modern teaching resource guides you through the **first steps in programming** using a little text and a lot of **code examples** and **video** explanations, combined with lots of carefully selected **practical problems** and a **judge system** for instant automatic code evaluation.

The teaching content is designed personally by **Dr. Svetlin Nakov**, who has helped in the last 20 years to more than **100 000 beginners** to start with programming and begin learning the **software engineering** profession.

Dr. Nakov is famous with his **inspiration** and ability to teach programming skills **gradually**, using **small learning units**, full of real world examples and **practical problem solving**.

Remember that **programming skills** can be learned only by a lot of **code writing** and a lot of **practical problem solving**, rather than by reading books and watching videos, so be sure that you **solve the exercises** in each book section. Good luck!

Web site: csharp-book.softuni.org



AUTHORS TEAM

Aleksander Krastev
Aleksander Lazarov
Angel Dimitriev
Vasko Viktorov
Ventsislav Petrov
Daniel Tsvetkov
Dimitar Tatarski
Dimo Dimov
Diyan Tonchev
Elena Rogleva
Zhivko Nedyalkov
Julieta Atanasova
Zahariya Pehlivanova
Ivelin Kirilov
Iskra Nikolova
Kalin Primov
Kristiyan Pamidov
Luboslav Lubenov
Nikolay Bankin
Nikolay Dimov
Pavlin Petkov
Petar Ivanov
Preslav Mihaylov
Rositsa Nenova
Ruslan Filipov
Svetlin Nakov
Stefka Vasileva
Teodor Kurtev
Tonyo Zhelev
Hristiyan Hristov
Hristo Hristov
Tsvetan Iliev
Yulian Linev
Yanitsa Vuleva

ISBN 978-619-00-0902-3



9 786190 009023 >