

СВЕТЛИН НАКОВ
& КОЛЕКТИВ



C#

**ОСНОВИ НА
ПРОГРАМИРАНЕТО
със **C#****

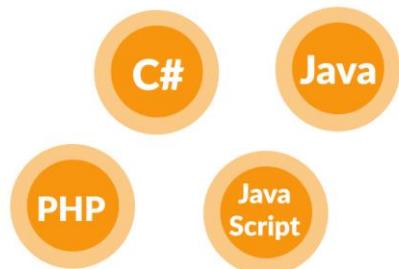
Кратко съдържание

| | |
|---|-----|
| Кратко съдържание | 3 |
| Съдържание | 7 |
| Предговор | 13 |
| Глава 1. Първи стъпки в програмирането | 27 |
| Глава 2.1. Прости пресмятания с числа..... | 61 |
| Глава 2.2. Прости пресмятания с числа – изпитни задачи..... | 93 |
| Глава 3.1. Прости проверки..... | 111 |
| Глава 3.2. Прости проверки – изпитни задачи..... | 141 |
| Глава 4.1. По-сложни проверки | 155 |
| Глава 4.2. По-сложни проверки – изпитни задачи..... | 185 |
| Глава 5.1. Повторения (цикли) | 207 |
| Глава 5.2. Повторения (цикли) – изпитни задачи | 231 |
| Глава 6.1. Вложени цикли..... | 249 |
| Глава 6.2. Вложени цикли – изпитни задачи | 269 |
| Глава 7.1. По-сложни цикли..... | 283 |
| Глава 7.2. По-сложни цикли – изпитни задачи | 315 |
| Глава 8.1. Подготовка за практически изпит – част I | 327 |
| Глава 8.2. Подготовка за практически изпит – част II | 353 |
| Глава 9.1. Задачи за шампиони – част I..... | 371 |
| Глава 9.2. Задачи за шампиони – част II..... | 385 |
| Глава 10. Методи | 401 |
| Глава 11. Хитрости и хакове | 433 |
| Заключение..... | 447 |

Качествено образование,
професия и работа за

Софтуерни инженери

- ✓ Безплатен старт за начинаещи - **присъствено и онлайн**
- ✓ Избор измежду **най-търсените професии** в софтуерната индустрия
- ✓ Съдействие за **кариерен старт**
- ✓ Отлични преподаватели, ментори и студентска общност



Софтууни предоставя съвременно иновативно образование за програмиране, ИТ и дигитални умения на хиляди млади хора. Програмата "**Софтуерен университет**" изгражда истински **професионалисти в света на програмирането**.

Включете се в цялостната програма по софтуерно инженерство на Софтууни, за да усвоите **най-търсените съвременни софтуерни технологии** чрез модерна учебна методика и с много практически задачи и проекти. **Учебният план** е разработен съвместно с ИТ фирмите и изгражда най-търсените от тях умения.

Софтууни работи пряко с компаниите от софтуерната индустрия, съдейтайки на своите студенти за реализацията им като успешни софтуерни инженери.

ПЪТЯТ НА СТУДЕНТА В СОФТУНИ



Кандидатствай

softuni.bg/apply

Основи на програмирането със C#

Светлин Наков и колектив

Александър Кръстев
Александър Лазаров
Ангел Димитриев
Васко Викторов
Венцислав Петров
Даниел Цветков
Димитър Татарски
Димо Димов
Диян Тончев
Елена Роглева
Живко Недялков
Жулиета Атанасова
Захария Пехливанова
Ивелин Кирилов
Искра Николова
Калин Примов
Кристиян Памидов

Любослав Любенов
Николай Банкин
Николай Димов
Павлин Петков
Петър Иванов
Росица Ненова
Руслан Филипов
Светлин Наков
Стефка Василева
Теодор Куртев
Тоньо Желев
Християн Христов
Христо Христов
Цветан Илиев
Юlian Линев
Яница Вълева

ISBN: 978-619-00-0635-0

София, 2017

Основи на програмирането със C#

© Светлин Наков и колектив, 2017 г.

Първо издание, май 2017 г.

Настоящата книга се разпространява **свободно** под **отворен лиценз CC-BY-SA**, който определя следните права и задължения:

- **Споделяне** – можете да копирате и разпространявате книгата свободно във всякакви формати и медиии.
- **Адаптиране** – можете да копирате, миксирате и променяте части от книгата и да създавате нови материали на базата на извадки от нея.
- **Признание** – при използване на извадки от книгата трябва да цитирате оригиналния източник, настоящия лиценз и да описете направените промени, без да въвеждате потребителя в заблуда, че оригиналните автори подкрепят вашата работа.
- **Подобно споделяне** – ако създавате материали чрез миксиране, промяна и копиране на извадки от книгата, трябва да споделите резултата под същия или подобен лиценз.

Всички запазени марки, използвани в тази книга, са собственост на техните притежатели.

Издателство: Фабер, гр. Велико Търново

ISBN: 978-619-00-0635-0

Корица: Марина Шидерова – <http://shideroff.com>

Официален уеб сайт: <https://csharp-book.softuni.bg>

Официална Facebook страница: <https://fb.com/IntroProgrammingBooks>

Сурс код: <https://github.com/SoftUni/Programming-Basics-Book-CSharp-BG>

Съдържание

| | |
|--|-----------|
| Кратко съдържание | 3 |
| Съдържание | 7 |
| Предговор | 13 |
| За кого е тази книга? | 13 |
| Защо избрахме езика C#? | 14 |
| Книгата на други програмни езици: Java, JavaScript, C++, Python, PHP, Ruby | 14 |
| Програмиране се учи с много писане, не с четене!..... | 14 |
| За Софтуерния университет (Софтууни)..... | 15 |
| Как се става програмист? | 17 |
| Книгата в помощ на учителите..... | 22 |
| Историята на тази книга | 22 |
| Официален сайт на книгата | 23 |
| Форум за вашите въпроси..... | 24 |
| Официална Facebook страница на книгата..... | 24 |
| Лиценз и разпространение | 24 |
| Докладване на грешки | 25 |
| Приятно четене!..... | 25 |
| Глава 1. Първи стъпки в програмирането | 27 |
| Видео | 27 |
| Какво означава "да програмираме"? | 27 |
| Как да напишем конзолна програма? | 31 |
| Пример: създаване на конзолна програма "Hello C#" | 36 |
| Тествайте програмите за свирене на ноти..... | 41 |
| Типични грешки в C# програмите | 41 |
| Какво научихме от тази глава? | 43 |
| Упражнения: първи стъпки в коденето | 43 |
| Конзолни, графични и уеб приложения | 48 |
| Глава 2.1. Прости пресмятания с числа..... | 61 |
| Видео | 61 |
| Системна конзола | 61 |
| Четене на числа от конзолата | 62 |
| Пресмятания в програмирането | 63 |
| Типове данни и променливи | 63 |
| Четене на дробно число от конзолата | 64 |
| Четене и печтане на текст | 64 |
| Съединяване на текст и числа | 65 |
| Аритметични операции | 66 |
| Съединяване на текст и число | 68 |
| Числени изрази | 69 |

| | |
|---|------------|
| Какво научихме от тази глава? | 71 |
| Упражнения: прости пресмятания..... | 71 |
| Графични приложения с числови изрази | 86 |
| Глава 2.2. Прости пресмятания с числа – изпитни задачи | 93 |
| Четене на числа от конзолата | 93 |
| Извеждане на текст по шаблон (placeholder)..... | 93 |
| Аритметични оператори..... | 93 |
| Конкатенация | 94 |
| Изпитни задачи | 94 |
| Задача: учебна зала..... | 94 |
| Задача: зеленчукова борса | 98 |
| Задача: ремонт на плочки | 100 |
| Задача: парички | 103 |
| Задача: дневна печалба | 107 |
| Глава 3.1. Прости проверки | 111 |
| Видео..... | 111 |
| Сравняване на числа..... | 111 |
| Прости проверки..... | 112 |
| Проверки с if-else конструкция | 113 |
| За къдрявите скоби { } след if / else..... | 113 |
| Живот на променлива | 115 |
| Серии от проверки..... | 116 |
| Упражнения: прости проверки | 117 |
| Дебъгване – прости операции с дебъгер..... | 121 |
| Упражнения: прости проверки | 123 |
| Графично (desktop) приложение | 135 |
| Глава 3.2. Прости проверки – изпитни задачи | 141 |
| Изпитни задачи | 141 |
| Задача: цена за транспорт | 141 |
| Задача: тръби в басейн..... | 144 |
| Задача: поспаливата котка Том | 146 |
| Задача: реколта | 149 |
| Задача: фирма | 151 |
| Глава 4.1. По-сложни проверки..... | 155 |
| Видео..... | 155 |
| Вложени проверки | 155 |
| По-сложни проверки..... | 157 |
| Логическо "ИЛИ" | 160 |
| Логическо отрицание..... | 162 |

| | |
|---|------------|
| Операторът скоби () | 162 |
| По-сложни логически условия | 162 |
| Условна конструкция switch-case | 166 |
| Какво научихме от тази глава? | 169 |
| Упражнения: по-сложни проверки | 170 |
| Упражнения: GUI с по-сложни проверки | 175 |
| Глава 4.2. По-сложни проверки – изпитни задачи..... | 185 |
| Вложени проверки..... | 185 |
| Switch-case проверки | 185 |
| Изпитни задачи | 186 |
| Задача: навреме за изпит | 186 |
| Задача: пътешествие..... | 190 |
| Задача: операции между числа..... | 194 |
| Задача: билети за мач..... | 198 |
| Задача: хотелска стая..... | 202 |
| Глава 5.1. Повторения (цикли)..... | 207 |
| Видео | 207 |
| Повторения на блокове код (for цикъл)..... | 207 |
| Code Snippet за for цикъл във Visual Studio | 209 |
| Какво научихме от тази глава? | 217 |
| Упражнения: повторения (цикли)..... | 217 |
| Упражнения: графични и уеб приложения | 221 |
| Глава 5.2. Повторения (цикли) – изпитни задачи | 231 |
| Изпитни задачи | 231 |
| Задача: хистограма | 231 |
| Задача: умната Лили | 236 |
| Задача: завръщане в миналото | 239 |
| Задача: болница..... | 242 |
| Задача: деление без остатък | 244 |
| Задача: логистика | 246 |
| Глава 6.1. Вложени цикли..... | 249 |
| Видео | 249 |
| Вложени цикли | 250 |
| Чертане на по-сложни фигури | 256 |
| Какво научихме от тази глава? | 262 |
| Упражнения: чертане на фигурки в уеб среда..... | 263 |
| Глава 6.2. Вложени цикли – изпитни задачи | 269 |
| Изпитни задачи | 269 |
| Задача: чертане на крепост | 269 |

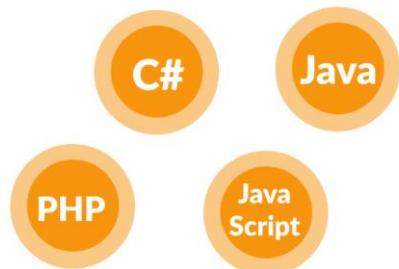
| | |
|--|------------|
| Задача: пеперуда..... | 271 |
| Задача: знак "Стоп" | 273 |
| Задача: стрелка..... | 276 |
| Задача: брадва | 278 |
| Глава 7.1. По-сложни цикли..... | 283 |
| Видео..... | 283 |
| Цикли със стъпка..... | 283 |
| While цикъл..... | 286 |
| Най-голям общ делител (НОД) | 287 |
| Алгоритъм на Евклид..... | 288 |
| Do-while цикъл | 289 |
| Безкрайни цикли и операторът break..... | 291 |
| Вложени цикли и операторът break..... | 295 |
| Справяне с грешни данни: try-catch | 296 |
| Задачи с цикли..... | 298 |
| Какво научихме от тази глава? | 303 |
| Упражнения: уеб приложения с по-сложни цикли..... | 304 |
| Глава 7.2. По-сложни цикли – изпитни задачи | 315 |
| Изпитни задачи | 315 |
| Задача: генератор за тъпи пароли | 315 |
| Задача: магически числа | 318 |
| Задача: спиращо число | 322 |
| Задача: специални числа | 323 |
| Задача: цифри | 324 |
| Глава 8.1. Подготовка за практически изпит – част I..... | 327 |
| Видео..... | 327 |
| Практически изпит по “Основи на програмирането” | 327 |
| Система за онлайн оценяване (Judge)..... | 327 |
| Задачи с прости пресмятания..... | 327 |
| Задачи с единична проверка | 331 |
| Задачи с по-сложни проверки | 335 |
| Задачи с единиччен цикъл | 339 |
| Задачи за чертане на фигурки на конзолата | 344 |
| Задачи с вложени цикли с по-сложна логика | 348 |
| Глава 8.2. Подготовка за практически изпит – част II..... | 353 |
| Видео..... | 353 |
| Изпитни задачи | 353 |
| Задача: разстояние | 353 |
| Задача: смяна на плочки | 357 |

| | |
|--|------------|
| Задача: магазин за цветя | 359 |
| Задача: оценки..... | 362 |
| Задача: коледна шапка | 364 |
| Задача: комбинации от букви..... | 367 |
| Глава 9.1. Задачи за шампиони – част I..... | 371 |
| По-сложни задачи върху изучавания материал..... | 371 |
| Задача: пресичащи се редици | 371 |
| Задача: магически дати | 376 |
| Задача: пет специални букви | 380 |
| Глава 9.2. Задачи за шампиони – част II..... | 385 |
| По-сложни задачи върху изучавания материал..... | 385 |
| Задача: дни за страстно пазаруване..... | 385 |
| Задача: числен израз | 390 |
| Задача: бикове и крави | 395 |
| Глава 10. Методи | 401 |
| Какво е "метод"? | 401 |
| Методи с параметри | 406 |
| Връщане на резултат от метод | 411 |
| Варианти на методи | 417 |
| Вложени методи (локални функции) | 420 |
| Именуване на методи и утвърдени практики | 422 |
| Какво научихме от тази глава? | 424 |
| Упражнения | 425 |
| Глава 11. Хитрости и хакове | 433 |
| Форматиране на кода | 433 |
| Именуване на елементите на кода..... | 435 |
| Бързи клавиши във Visual Studio..... | 436 |
| Шаблони с код (code snippets)..... | 437 |
| Техники за дебъгване на кода | 441 |
| Справочник с хитрости | 443 |
| Какво научихме от тази глава? | 446 |
| Заключение..... | 447 |
| Тази книга е само първа стъпка!..... | 447 |
| Накъде да продължим след тази книга? | 448 |
| Онлайн общности за стартиращите в програмирането | 451 |
| Успех на всички! | 452 |

Качествено образование,
професия и работа за

Софтуерни инженери

- ✓ Безплатен старт за начинаещи - **присъствено и онлайн**
- ✓ Избор измежду **най-търсените професии** в софтуерната индустрия
- ✓ Съдействие за **кариерен старт**
- ✓ Отлични преподаватели, ментори и студентска общност



Софтууни предоставя съвременно иновативно образование за програмиране, ИТ и дигитални умения на хиляди млади хора. Програмата "**Софтуерен университет**" изгражда истински **професионалисти в света на програмирането**.

Включете се в цялостната програма по софтуерно инженерство на Софтууни, за да усвоите **най-търсените съвременни софтуерни технологии** чрез модерна учебна методика и с много практически задачи и проекти. **Учебният план** е разработен съвместно с ИТ фирмите и изгражда най-търсените от тях умения.

Софтууни работи пряко с компаниите от софтуерната индустрия, съдейтайки на своите студенти за реализацията им като успешни софтуерни инженери.

ПЪТЯТ НА СТУДЕНТА В СОФТУНИ



Кандидатствай

softuni.bg/apply

Предговор

Книгата "Основи на програмирането" е официален учебник за курса "Programming Basics" в Софтуерния университет (Софтуни):

<https://softuni.bg/courses/programming-basics>

Тя запознава читателите с писането на **програмен код** на начално ниво (basic coding skills), работа със **среда за разработка** (IDE), използване на **променливи** и **данни**, **оператори и изрази**, работа с **конзолата** (четене на входни данни и печтане на резултати), използване на **условни конструкции** (**if, if-else, switch-case**), **цикли** (**for, while, do-while, foreach**) и работа с **методи** (деклариране и извикване на методи, подаване на параметри и връщане на стойност). Използват се езикът за програмиране **C#** и средата за разработка **Visual Studio**. Обхванатият учебен материал дава базова подготовка за по-задълбочено изучаване на програмирането и подготвя читателите за приемния изпит в Софтуни.



Тази книга ви дава само **първите стъпки към програмирането**. Тя обхваща съвсем начални умения, които предстои да развивате години наред, докато достигнете до ниво, достатъчно за започване на работа като програмист.

Книгата се използва и като неофициален учебник за училищните курсове по програмиране в професионалните гимназии, изучаващи професиите "Програмист", "Приложен програмист" и "Системен програмист", както и като допълнително учебно пособие в началните курсове по програмиране в **средните училища, профилираните и математическите гимназии**, за паралелките с профил "информатика и информационни технологии".

За кого е тази книга?

Тази книга е подходяща за **напълно начинаещи в програмирането**, които искат да опитат какво е да програмираш и да научат основните конструкции за създаване на програмен код, които се използват в софтуерната разработка, независимо от езиците за програмиране и използваните технологии. Книгата дава една **солидна основа** от практически умения, които се използват за по-нататъшно обучение в програмирането и разработката на софтуер.

За всички, които не са преминали **бесплатния курс по основи на програмирането за напълно начинаещи в Софтуни** (<https://softuni.bg/apply>), специално препоръчваме да го запишат **напълно бесплатно**, защото програмиране се учи с правене, не с четене! На курса ще получите бесплатно достъп до учебни занятия, обяснения и демонстрации на живо или онлайн (като видео уроци), **много практика и писане на код**, помош при решаване на задачите след всяка тема, достъп до преподаватели, асистенти и ментори, както и форум и дискусионни групи за въпроси, достъп до общност от хиляди навлизящи в програмирането и всяка друга помощ за начинаещия.

Безплатният курс в СофтУни за напълно начинаещи е подходящ за **ученици** (от 5 клас нагоре), **студенти и работещи** други професии, които искат да натрупат технически знания и да разберат дали им харесва да програмират и дали биха се занимавали сериозно с разработка на софтуер за напред.

Нова група започва всеки месец. Курсът "Programming Basics" в СофтУни се организира регулярно с няколко различни езика за програмиране, така че опитайте. Обучението е **бесплатно** и може да се откажете по всяко време, ако не ви допадне. **Записването** за бесплатно **присъствено** или **онлайн** обучение за стартсиращи в програмирането е достъпно през формуляра за кандидатстване в СофтУни: <https://softuni.bg/apply>.

Защо избрахме езика C#?

За настоящата книга избрахме езика **C#**, защото е **съвременен език** за програмиране от високо ниво с отворен код и същевременно е лесен за научаване и **подходящ за начинаещи**. Като употреба C# е **широкоразпространен**, с добре развита екосистема, с многобройни библиотеки и технологични рамки и съответно дава много **перспективи** за развитие. C# комбинира парадигмите на процедурното, обектно-ориентираното и функционалното програмиране по съвременен начин с лесен за употреба синтаксис. В книгата ще използваме **езика C#** и средата за разработка **Visual Studio**, които са достъпни бесплатно от Microsoft.

Както ще обясним по-късно, езикът за програмиране, с който стартсираме, няма съществено значение, но все пак трябва да ползваме някакъв програмен език, и в тази книга сме избрали именно **C#**. Книгата може да се намери преведена огледално и на други езици за програмиране като **Java** и **JavaScript** (вж. <https://csharp-book.softuni.bg>).

Книгата на други програмни езици: Java, JavaScript, C++, Python, PHP, Ruby

Настоящата книга е достъпна на няколко езика за програмиране (или е в процес на адаптация за тях):

- Основи на програмирането със C#: <https://csharp-book.softuni.bg>
- Основи на програмирането с Java: <https://java-book.softuni.bg>
- Основи на програмирането с JavaScript: <https://js-book.softuni.bg>
- Основи на програмирането с Python: <https://python-book.softuni.bg>
- Основи на програмирането със C++: <https://cpp-book.softuni.bg>
- Основи на програмирането с PHP: <https://php-book.softuni.bg>

Ако предпочитате друг език за програмиране, изберете си от списъка по-горе.

Програмиране се учи с много писане, не с четене!

Ако някой си мисли, че ще прочете една книга и ще се научи да програмира без да пише код и да решава здраво задачи, определено е в заблуда. Програмирането се учи с **много, много практика**, с писане на код всеки ден и решаване на стотици, дори хиляди задачи, сериозно и с постоянство, в продължение на години.

Трябва **да решавате задачи усърдно**, да бъркате, да се поправяте, да търсите решения и информация в Интернет, да пробвате, да експериментирате, да намирате по-добри решения, да свиквате с кода, със синтаксиса, с езика за програмиране, със средата за разработка, с търсенето на грешки и дебъгването на неработещ код, с разсъжденията над задачите, с алгоритмичното мислене, с разбиването на проблемите на стъпки и имплементацията на всяка стъпка, да трупате опит и да вдигате уменията си всеки ден, защото да се научиш да пишеш код е само **първата стъпка към професията "софтуерен инженер"**. Имате да учите много, наистина много!

Съветваме читателя като минимум **да пробва всички примери от книгата**, да си поиграе с тях, да ги променя и пробва. Още по-важни от примерите, са **задачите за упражнения**, защото те развиват практическите умения на програмиста.

Решавайте всички задачи от книгата, защото програмиране се учи с практика! Задачите след всяка тема са внимателно подбрани, така че да покриват в дълбочина обхванатия учебен материал, а целта на решаването на всички задачи от всички обхванати теми е да дадат **цялостни умения за писане на програмен код** на начално ниво (каквато е целта и на тази книга). На курсовете в СофтУни не случайно **наблягаме на практиката** и решаването на задачи, и в някои курсове писането на код в клас е над 70% от целия курс.



Решавайте всички задачи за упражнения от книгата. Иначе нищо няма да научите! Програмиране се учи с писане на много код и решаване на хиляди задачи!

За Софтуерния университет (Софтууни)

Софтуерният университет (Софтууни) е най-мащабният учебен център за софтуерни инженери в България (вж. <https://softuni.bg>). През него преминават десетки хиляди студенти всяка година. СофтУни отваря врати през 2014 г. като продължение на усилията на д-р Светлин Наков (<http://nakov.com>) масирано да изгражда **кадърни софтуерни специалисти** чрез истинско, съвременно и качествено образование, което комбинира фундаментални знания със съвременни софтуерни технологии и много практика.

Софтуерният университет предоставя **качествено образование, професия, работа и възможност за придобиване на бакалавърска степен** за програмисти, софтуерни инженери и ИТ специалисти. СофтУни изгражда изключително успешно трайна връзка между **образование и индустрия**, като си сътрудничи със стотици софтуерни фирми, осигурява работа и стажове на своите студенти, предоставя

качествени специалисти за софтуерната индустрия и директно отговаря на нуждите на работодателите чрез учебния процес.

Безплатните курсове по програмиране в СофтУни

Софтуерният университет (Софтуни) организира **безплатни курсове по програмиране** за **напълно начинаещи** в цяла България - присъствено и онлайн. Целта е **всеки, който има интерес** към програмиране и технологии, **да опита програмирането** и да се увери сам дали то е интересно за него и дали иска да се занимава сериозно с разработка на софтуер. Можете да се запишете за **бесплатния курс по основи на програмирането** от страницата за кандидатстване в СофтУни:

<https://softuni.bg/apply>

Безплатните курсове по основи на програмирането в СофтУни имат за цел да ви запознаят с **основните логически концепции** от света на софтуерната разработка, които ще можете да приложите при всеки един език за програмиране. Те включват **типове данни, променливи, условни конструкции, цикли, методи** и други похвати за изграждане на програмна логика. Обученията са **изключително практически насочени**, което означава, че **силно се набляга на упражнения**, а вие получавате възможността да приложите знанията си още докато ги усвоявате.

Настоящият **учебник по програмиране** съществува безплатните курсове по програмиране за начинаещи в СофтУни и служи като допълнително учебно помагало, в помощ на учебния процес.

Judge системата за проверка на задачите

Софтуни Judge системата (<https://judge.softuni.bg>) представлява автоматизирана система в Интернет за проверка на решения на задачи по програмиране чрез **поредица от тестове**. Предаването и проверката на задачите се извършва в **реално време**: пращаш решение и след секунди получаваш отговор дали е вярно. Всеки **успешно** преминат тест дава предвидените за него точки. При вярно решение получавате всички точки за задачата. При частично вярно решение получавате част от точките за дадената задача. При напълно грешно решение, получавате 0 точки.

Всички задачи от настоящата книга са достъпни за тестване в СофтУни judge и силно препоръчваме да ги тествате след като ги решите, за да знаете дали не изпускате нещо и дали наистина решението ви работи правилно, според изискванията на задачата.

Имайте предвид и някои **особености на SoftUni judge**:

- За всяка задача **judge** системата пази най-високия постигнат резултат. Ако качите решение с грешен код или по-слаб резултат от предишното ви изпратено, системата няма да ви отнеме точки.
- Изходните резултати на вашата програма се **сравняват** от системата стриктно с очаквания резултат. Всеки **излишен символ, липсваща запетайка или**

интервал може доведе до 0 точки на съответния тест. **Изходът**, който judge системата очаква, е описан в условието на всяка задача и към него не трябва да се добавя нищо повече.

Пример: ако в изхода се изисква да се отпечата число (напр. 25), не извеждайте описателни съобщения като **The result is: 25**, а отпечатайте точно каквото се изисква, т.е. само числото.

Софтуни judge системата е **достъпна по всяко време** от нейния сайт:

<https://judge.softuni.bg>

- За вход използвайте автентификацията си от сайта на Софтуни: softuni.bg.
- Използването на системата е **безплатно** и не е обвързано с участието в курсовете на Софтуни.

Убедени сме, че след няколко изпратени задачи, ще ви хареса да получавате **ментална обратна връзка** дали написаното от вас решение е вярно, и judge система ще ви стане най-любимия помощник при учене на програмирането.

Как се става програмист?

Драги читатели, сигурно много от вас имат амбицията да стават програмисти, да си изкарват прехраната с разработка на софтуер или да работят в ИТ сектора. Затова сме приготвили за вас **кратко ръководство "Как се става програмист"**, за да ви ориентираме за стъпките към тази така желана професия.

Програмист (на ниво започване на работа в софтуерна фирма) се става за **най-малко 1-2 години здраво учене и писане на код всеки ден**, решаване на няколко хиляди задачи по програмиране, разработка на няколко по-сериозни практически проекта и трупане на много опит с писането на код и разработката на софтуер. Не става за един месец, нито за два! Професията на софтуерния инженер изисква голям обем познания, покрити с много практика.

Има **4 основни групи умения**, които всички програмисти трябва да притежават. Повечето от тези умения са устойчиви във времето и не се влияят съществено от развитието на конкретните технологии (които се променят постоянно). Това са умения които **всеки добър програмист притежава** и към които всеки новобранец трябва да се стреми.

Умение #1 – кодене (20%)

Да се научите **да пишете код** формира около 20% от минималните умения на програмиста, необходими за започване на работа в тази сфера. Умението да кодиш включва следните компоненти:

- работа с променливи, проверки, цикли
- ползване на функции, методи, класове и обекти
- работа с данни: масиви, списъци, хеш-таблици, стрингове

Умението да кодиш **може да се усвои за няколко месеца** усилено учене и здраво решаване на практически задачи с писане на код всеки ден. Настоящата книга покрива само първата точка от умението да кодиш: **работка с променливи, проверки и цикли**. Останалото остава да се научи в последващи обучения, курсове и книги.

Книгата (и курсовете, базирани на нея) дават само началото от едно дълго и сериозно учене, по пътя на професионалното програмиране. Ако не усвоите учебния материал от настоящата книга, няма как да станете програмист. Ще ви липсват фундаментални основи и ще ви става все по-трудно напред. Затова **отделете достатъчно внимание на основите на програмирането**: решавайте здраво задачи и пишете много код месеци наред, докато се научите **да решавате с лекота всички задачи от тази книга**. Тогава продължете напред.

Обръщаме внимание, че **езикът за програмиране няма съществено значение** за умението да кодиш. Или можеш да кодиш или не. Ако можеш да кодиш на C#, лесно ще се научиш да кодиш и на Java, и на C++, и на друг език. Затова **уменията да кодираш** се изучават доста сериозно в началните курсове за софтуерни инженери в СофтУни (вж. учебния план от <https://softuni.bg/curriculum>) и с тях стартира всяка книга за програмиране за напълно начинаещи, включително нашата.

Умение #2 – алгоритмично мислене (30%)

Алгоритмичното (логическо, инженерно, математическо, абстрактно) мислене формира около 30% от минималните умения на програмиста за старт в професията. **Алгоритмичното мислене** е умението да разбивате една задача на логическа последователност от стъпки (алгоритъм) и да намирате решение за всяка отделна стъпка, след което да сглобявате стъпките в работещо решение на първоначалната задача. Това е най-важното умение на програмиста.

Как да си изградим алгоритмично мислене?

- Алгоритмичното мислене се развива се чрез решаване на **много (1000+)** задачи по програмиране, възможно най-разнообразни. Това е рецептата: решаване на хиляди практически задачи, измисляне на алгоритъм за тях и имплементиране на алгоритъма, заедно с дебъгване на грешките по пътя.
- Помагат физика, математика и/или подобни науки, но не са задължителни! Хората с **инженерни и технически наклонности** обикновено по-лесно се научават да мислят логически, защото имат вече изградени умения за решаване на проблеми, макар и не алгоритмични.
- Способността **да решавате задачи по програмиране** (за която е нужно алгоритмично мислене) е изключително важна за програмиста. Много фирми изпитват единствено това умение при интервюта за работа.

Настоящата книга развива **начално ниво на алгоритмично мислене**, но съвсем не е достатъчна, за да ви направи добър програмист. За да станете кадърни в професията, ще трябва да добавите **умения за логическо мислене и решаване на задачи** отвъд обхвата на тази книга, например работа със **структурни данни** (масиви,

списъци, матрици, хеш-таблици, дървовидни структури) и базови **алгоритми** (търсение, сортиране, обхождане на дървовидни структури, рекурсия и други).

Умения за алгоритмично мислене се развиват сериозно в началните курсове за софтуерни инженери в СофтУни (вж. учебния план от <http://softuni.bg/curriculum>), както и в специализираните курсове по **структурни от данни и алгоритми**.

Както може би се досещате, **езикът за програмиране няма значение** за развиващото се на алгоритмичното мислене. Да мислиш логически е универсално, дори не е свързано само с програмирането. Именно заради силно развитото логическото мислене се счита, че **програмистите са доста умни** и че прост човек не може да стане програмист.

Умение #3 – фундаментални знания за професията (25%)

Фундаментални знания и умения за програмирането, разработката на софтуер, софтуерното инженерство и компютърните науки формират около 25% от минималните умения на програмиста за започване на работа. Ето по-важните от тези **фундаментални знания и умения**:

- **базови математически концепции**, свързани с програмирането: координатни системи, вектори и матрици, дискретни и недискретни математически функции, крайни автомати и state machines, понятия от комбинаториката и статистика, сложност на алгоритъм, математическо моделиране и други
- **структури от данни и алгоритми** - списъци, дървета, хеш-таблици, графи, търсение, сортиране, рекурсия, обхождане и други
- **обектно-ориентирано програмиране (ООП)** – работа с класове, обекти, наследяване, абстракция, интерфейси, капсуляция на данни, полиморфизъм, управление на изключения, шаблони за дизайн
- **функционално програмиране (ФП)** - работа с ламбда функции, функции от по-висок ред, функции, които връщат като резултат функция, затваряне на състояние във функция (closure) и други
- **бази данни** - релационни и нерелационни бази данни, моделиране на бази данни (таблици и връзки между тях), език за заявки SQL, технологии за обектно-релационен достъп до данни (ORM), транзакционност и управление на транзакции
- **мрежово програмиране** - мрежови протоколи, мрежова комуникация, TCP / IP, понятия, инструменти и технологии от компютърните мрежи
- взаимодействие **клиент-сървър**, комуникация между системи, back-end технологии, front-end технологии, MVC архитектури
- **технологии за сървърна (back-end) разработка** - архитектура на уеб сървър, HTTP протокол, MVC архитектура, REST архитектура, templating engines
- **уеб front-end технологии** - HTML, CSS, JS, HTTP, DOM, AJAX, комуникация с back-end, извикване на REST API

- **паралелно програмиране и асинхронност** - управление на нишки, асинхронни задачи, promises, общи ресурси и синхронизация на достъпа
- **вградени системи** - микроконтролери, управление на периферия, управление на цифров и аналогов вход и изход, достъп до сензори,
- **софтуерно инженерство** – софт контрол системи, управление на разработката, методологии за софтуерна разработка, софтуерни изисквания и прототипи, софтуерен дизайн, софтуерни архитектури, софтуерна документация
- **софтуерно тестване** – компонентно тестване (unit testing), test-driven development, QA инженерство, докладване на грешки и трекери за грешки, автоматизация на тестването, билд процеси и непрекъсната интеграция

Трябва да поясним и този път, че **езикът за програмиране няма значение** за усвояването на всички тези умения. Те се натрупват бавно, в течение на много години практика в професията. Някои знания са фундаментални и могат да се усвояват теоретично, но за пълното им разбиране и осъзнаването им дълбочина, са необходими години практика.

Фундаментални знания и умения за програмирането, разработката на софтуер, софтуерното инженерство и компютърните науки се учат по време на **цялостната програма за софтуерни инженери в СофтУни** (<https://softuni.bg/curriculum>), както и с редица **изборни курсове** (<https://softuni.bg/trainings/opencourses>). Работата с разнообразни софтуерни библиотеки, програмни интерфейси (APIs), технологични рамки (frameworks) и софтуерни технологии и тяхното взаимодействие, постепенно изграждат тези знания и умения, така че не очаквайте да ги добиете от единичен курс, книга или проект.

За започване на работа като програмист обикновено са достатъчни само **начални познания в изброените по-горе области**, а задълбоването става на работното място според използваните технологии и инструменти за разработка в съответната фирма и екип.

Умение #4 – езици за програмиране и софтуерни технологии (25%)

Езиците за програмиране и технологиите за софтуерна разработка формират около 25% от минималните умения на програмиста. Те са най-обемни за усвояване, но най-бързо се променят с времето. Ако погледнем **обявите за работа** от софтуерната индустрия, там често се споменават всякакви думички (като изброените по-долу), но всъщност в обявите мълчаливо **се подразбират първите три умения**: да кодиш, да мислиш алгоритмично и да владееш фундамента на компютърните науки и софтуерното инженерство.

За тези чисто технологични умения вече **езикът за програмиране има значение**.

- **Обърнете внимание:** само за тези 25% от професията има значение езикът за програмиране!

- За останалите 75% от уменията няма значение езикът и тези умения са устойчиви във времето и преносими между различните езици и технологии.

Ето и някои често използвани **езици и технологии** (software development stacks), които се търсят от софтуерните фирми (актуални през 2017 г.):

- **C#** + ООП, ФП, класовете от .NET + база данни SQL Server, Entity Framework, ASP.NET MVC, HTTP + HTML + CSS + JS + DOM + jQuery
- **Java** + Java API classes + ООП + ФП + бази данни + MySQL + HTTP, уеб програмиране + HTML + CSS + JS + DOM + jQuery + JSP/Servlets + Spring MVC или Java EE / JSF
- **PHP** + ООП + бази данни + MySQL + HTTP, уеб програмиране + HTML + CSS + JS + DOM + jQuery + Laravel / Symfony / друг MVC framework за PHP
- **JavaScript (JS)** + ООП + ФП + бази данни + MongoDB, MySQL + HTTP, уеб програмиране + HTML + CSS + JS + DOM + jQuery + Node.js + Express + Angular + React
- **Python** + ООП + ФП + бази данни + MongoDB, MySQL + HTTP, уеб програмиране + HTML + CSS + JS + DOM + jQuery + Django
- **C++** + ООП + STL + Boost + native development + бази данни + HTTP + други езици
- **Swift** + MacOS + iOS, Cocoa Touch + XCode + HTTP + REST + други езици

Ако изброените по-горе думички ви изглеждат страшни и абсолютно непонятни, значи сте съвсем в началото на кариерата си и имате **да учите още години** докато достигнете професията "софтуерен инженер". Не се притеснявайте, всеки програмист преминава през един или няколко технологични стека и се налага да изучи **съвкупност от взаимосвързани технологии**, но в основата на всичко това стои **умението да пишеш програмна логика (да кодиш)**, което се развива в тази книга, и **умението да мислиш алгоритично (да решаваш задачи по програмиране)**. Без тях не може!

Езикът за програмиране няма значение

Както вече стана ясно, **разликата между езиците за програмиране** и по-точно между уменията на програмистите на различните езици и технологии, е в около **10-20% от уменията**.

- Всички програмисти имат около **80-90% еднакви умения**, които не зависят от езика! Това са уменията да програмираш и да разработваш софтуер, които са много подобни в различните езици за програмиране и технологии за разработка.
- Колкото повече езици и технологии владеете, толкова по-бързо ще учите нови и толкова по-малко ще усещате разлика между тях.

Наистина, **езикът за програмиране няма съществено значение**, просто трябва да се научите да програмирате, а това започва с **коденето** (настоящата книга),

продължава в по-сложни **концепции от програмирането** (като структури от, алгоритми, данни, ООП и ФП) и включва усвояване на **фундаментални знания и умения за разработката на софтуер, софтуерното инженерство и компютърните науки**.

Едва накрая, когато захванете конкретни технологии в даден софтуерен проект, ще ви трябват **конкретен език за програмиране**, познания за конкретни програмни библиотеки (APIs), работни рамки (frameworks) и софтуерни технологии (front-end UI технологии, back-end технологии, ORM технологии и други). Спокойно, ще ги научите, всички програмисти ги научават, но първо се научават на фундамента: **да програмират и то добре**.

Настоящата книга използва **езика за програмиране C#**, но той не е съществен и може да се замени с Java, JavaScript, Python, PHP, C++, Ruby, Swift, Go, Kotlin или друг език. За овладяване на **професията "софтуерен разработчик"** е необходимо да се научите да **кодите** (20%), да се научите да **мислите алгоритично** и да **решавате проблеми** (30%), да имате **фундаментални знания по програмиране и компютърни науки** (25%) и да владеете **конкретен език за програмиране и технологиите около него** (25%). Имайте търпение, за година-две всичко това може да се овладее на добро начално ниво, стига да сте сериозни и усърдни.

Книгата в помощ на учителите

Ако сте **учител по програмиране**, информатика или информационни технологии или искате **да преподавате програмиране**, тази книга ви дава нещо повече от добре структуриран учебен материал с много примери и задачи. **Бесплатно** към книгата получавате **качествено учебно съдържание** за преподаване в училище, на **български език**, съобразено с училищните изисквания:

- Учебни презентации (PowerPoint слайдове) за всяка една учебна тема, съобразени с 45-минутните часове в училищата – бесплатно.
- Добре разработени задачи за упражнения в клас и за домашно, с детайлно описани условия и примерен вход и изход – бесплатно.
- Система за автоматизирана проверка на задачите и домашните (online judge system), която да се използва от учениците, също бесплатно.
- Видео-уроци с методически указания от **бесплатния курс за учители по програмиране**, който се провежда регулярно от СофтУни фондацията.

Всички тези **бесплатни преподавателски ресурси** можете да намерите на сайта на СофтУни фондацията, заедно с учебно съдържание за цяла поредица от курсове по програмиране и софтуерни технологии. Изтеглете ги свободно от тук:

<http://softuni.foundation/projects/applied-software-developer-profession>

Историята на тази книга

Главен двигател и ръководител на проекта за създаване на настоящата **свободна книга по програмиране за начинаещи** с отворен код е **д-р Светлин Наков**. Той е

основен идеолог и създател на учебното съдържание от курса "Основи на програмирането" в СофтУни, който е използван за основа на книгата.

Всичко започва с масовите **безплатни курсове по основи на програмирането**, провеждани в цялата страна от 2014 г. насам, когато стартира инициативата "Софтуни". В началото тези курсове имат по-голям обхват и включват повече теория, но през 2016 г. д-р Светлин Наков изцяло ги преработва, обновява, опростява и насочва много силно към практиката. Така е създадено ядрото на **учебното съдържание от тази книга**.

Безплатните обучения на СофтУни за старт в програмирането са може би най-мащабните, провеждани някога в България. До 2017 г. курсът на СофтУни по основи на програмирането се провежда над 150 пъти в близо 40 български града присъствено и многократно онлайн, с над 50 000 участника. Съвсем естествено възниква и нуждата да се напише **учебник** за десетките хиляди участници в курсовете на СофтУни по програмиране за начинаещи. На принципа на свободния софтуер и свободното знание, Светлин Наков повежда **екип от доброволци** и задвижва този open-source проект, първоначално за създаване на книга по основи на програмирането с езика C#, а по-късно и с други езици за програмиране.

Проектът е част от организираните усилия на **Фондация "Софтуерен университет"** (<http://softuni.foundation>) да създава и разпространява отворено учебно съдържание за обучение на софтуерни инженери и ИТ специалисти.

Авторски колектив

Настоящата книга е разработена от широк авторски колектив от **доброволци**, които отделиха от своето време, за да ви подарят тези систематизирани знания и насоки при старта в програмирането. Списък на всички автори и редактори (по азбучен ред):

Александър Кръстев, Александър Лазаров, Ангел Димитриев, Васко Викторов, Венцислав Петров, Даниел Цветков, Димитър Татарски, Димо Димов, Диян Тончев, Елена Роглева, Живко Недялков, Жулиета Атанасова, Захария Пехливанова, Ивелин Кирилов, Искра Николова, Калин Примов, Кристиян Памидов, Любослав Любенов, Николай Банкин, Николай Димов, Павлин Петков, Петър Иванов, Росица Ненова, Руслан Филипов, Светлин Наков, Стефка Василева, Теодор Куртев, Тоньо Желев, Християн Христов, Христо Христов, Цветан Илиев, Юlian Линев, Яница Вълева

Официален сайт на книгата

Настоящата книга по **основи на програмирането със C#** за начинаещи е достъпна за свободно ползване в Интернет от адрес:

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

Това е **официалният сайт на книгата** и там ще бъде качвана нейната последна версия. Книгата е преведена огледално и на други езици за програмиране, посочени на нейния сайт.

Форум за вашите въпроси

Задавайте вашите въпроси към настоящата книга по основи на програмирането във форума на СофтУни:

<https://softuni.bg/forum>

В този дискусионен форум ще получите бесплатно **адекватен отговор по всяка въпроси от учебното съдържание на настоящия учебник**, както и други въпроси от програмирането. Общността на СофтУни за навлизящи в програмирането е толкова голяма, че обикновено отговор на зададен въпрос се получава **до няколко минути**. Преподавателите, асистентите и менторите от СофтУни също отговарят постоянно на вашите въпроси.

Поради големия брой учащи по настоящия учебник, във форума можете да намерите **решение на практически всяка задача от него**, споделено от ваш колега. Хиляди студенти преди вас вече са решавали същите задачи, така че ако закъсвате, потърсете из форума. Макар и задачите в курса "Основи на програмирането" да се сменят от време на време, споделянето е винаги настърчавано в СофтУни и затова лесно ще намерите решения и насоки за всички задачи.

Ако все пак имате конкретен въпрос, например защо не тръгва дадена програма, над която умувате от няколко часа, **задайте го във форума** и ще получите отговор. Ще се убудите колко добронамерени и отзивчиви са обитателите на СофтУни форума.

Официална Facebook страница на книгата

Книгата си има и **официална Facebook страница**, от която може да следите за новини около книгите от поредицата "Основи на програмирането", нови издания, събития и инициативи:

<https://fb.com/IntroProgrammingBooks>

Лиценз и разпространение

Книгата се разпространява **бесплатно** в електронен формат под отворен лиценз **CC-BY-SA** (<https://creativecommons.org/licenses/by-sa/4.0/>).

Книгата се издава и разпространява **на хартия** от СофтУни и хартиено копие може да се закупи от рецепцията на СофтУни (вж. <https://softuni.bg/contacts>).

Сурс кодът на книгата може да се намери в GitHub: <https://github.com/SoftUni/Programming-Basics-Book-CSharp-BG>.

Международен стандартен номер на книга ISBN: 978-619-00-0635-0.

Докладване на грешки

Ако откриете **грешки**, неточности или дефекти в книгата, можете да ги докладвате в официалния тракер на проекта:

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

Не обещаваме, че ще поправим всичко, което ни изпратите, но пък имаме желание **постоянно да подобряваме качеството** на настоящата книга, така че докладваните безспорни грешки и всички разумни предложения ще бъдат разгледани.

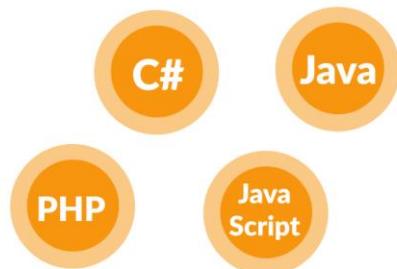
Приятно четене!

И не забравяйте **да пишете код** в големи количества, да пробвате **примерите** от всяка тема и най-вече да решавате **задачите от упражненията**. Само с четене няма да се научите да програмирате, така че решавайте задачи здраво!

Качествено образование,
професия и работа за

Софтуерни инженери

- ✓ Безплатен старт за начинаещи - **присъствено и онлайн**
- ✓ Избор измежду **най-търсените професии** в софтуерната индустрия
- ✓ Съдействие за **кариерен старт**
- ✓ Отлични преподаватели, ментори и студентска общност



Софтууни предоставя съвременно иновативно образование за програмиране, ИТ и дигитални умения на хиляди млади хора. Програмата "**Софтуерен университет**" изгражда истински **професионалисти в света на програмирането**.

Включете се в цялостната програма по софтуерно инженерство на Софтууни, за да усвоите **най-търсените съвременни софтуерни технологии** чрез модерна учебна методика и с много практически задачи и проекти. **Учебният план** е разработен съвместно с ИТ фирмите и изгражда най-търсените от тях умения.

Софтууни работи пряко с компаниите от софтуерната индустрия, съдейтайки на своите студенти за реализацията им като успешни софтуерни инженери.

ПЪТЯТ НА СТУДЕНТА В СОФТУНИ



Programming Basics

Кариерен Старт

Дипломиране

70/100 credits

80/100/150 credits

Кандидатствай

softuni.bg/apply

Глава 1. Първи стъпки в програмирането

В тази глава ще разберем какво е програмирането в неговата същина. Ще се запознаем с идеята за **програмни езици** и ще разгледаме **средите за разработка на софтуер** (IDE) и как да работим с тях, в частност с **Visual Studio**. Ще напишем и изпълним **първата си програма** на програмния език **C#**, а след това ще се упражним с няколко задачи: ще създадем конзолна програма, графично приложение и уеб приложение. Ще се научим как да проверяваме за коректност решението на задачите от тази книга в **Judge системата на СофтУни** и накрая ще се запознаем с типичните грешки, които често се допускат при писането на код и как да се предпазим от тях.

Видео

Гледайте видео урок по тази глава: <https://youtube.com/watch?v=LgT10WCw0M>.

Какво означава "да програмираме"?

Да програмираме означава да даваме команди на компютъра какво да прави, например "да иззвири някакъв звук", "да отпечатва нещо на екрана" или "да умножи две числа". Когато командите са няколко една след друга, те се наричат **компютърна програма**. Текстът на компютърните програми се нарича **програмен код** (или **сурс код** или за по-кратко **код**).

Компютърни програми

Компютърните програми представляват **поредица от команди**, които се изписват на предварително избран **език за програмиране**, например C#, Java, JavaScript, Python, Ruby, PHP, C, C++, Swift, Go или друг. За да пишем команди, трябва да знаем **синтаксиса и семантиката на езика**, с който ще работим, в нашия случай **C#**. Затова ще се запознаем със синтаксиса и семантиката на езика C# и с програмирането като цяло в настоящата книга, изучавайки стъпка по стъпка писането на код, от по-простите към по-сложните програмни конструкции.

Алгоритми

Компютърните програми обикновено изпълняват някакъв алгоритъм. **Алгоритмите** са последователност от стъпки, необходими за да се свърши определена работа и да се постигне някакъв очакван резултат, нещо като "рецепта". Например, ако пържим яйца, ние изпълняваме някаква рецепта (алгоритъм): загряваме мазнина в някакъв съд, чупим яйцата, изчакваме докато се изпържат, отместваме от огъня. Аналогично, в програмирането **компютърните програми изпълняват алгоритми**: поредица от команди, необходими, за да се свърши определена работа. Например, за да се подредят поредица от числа в нарастващ ред, е необходим алгоритъм, примерно да се намери най-малкото число и да се отпечата, от останалите числа да се намери отново най-малкото число и да се отпечата и това се повтаря докато числата свършат.

За удобство при създаването на програми, за писане на програмен код (команди), за изпълнение на програмите и за други операции, свързани с програмирането, ни е необходима и среда за разработка, например Visual Studio.

Езици за програмиране, компилатори, интерпретатори и среди за разработка

Езикът за програмиране е изкуствен език (синтаксис за изразяване), предназначен за задаване на **команди**, които искаме компютърът да прочете, обработи и изпълни. Чрез езиците за програмиране пишем поредици от команди (**програми**), които **задават какво да прави компютъра**. Изпълнението на компютърните програми може да се реализира с **компилатор** или с **интерпретатор**.

Компилаторът превежда кода от програмен език на **машинен код**, като за всяка от конструкциите (командите) в кода избира подходящ, предварително подготвен фрагмент от машинен код, като междувременно **проверява за грешки текста на програмата**. Заедно компилираните фрагменти съставят програмата в машинен код, както я очаква микропроцесорът на компютъра. След като е компилирана програмата, тя може да бъде директно изпълнена от микропроцесора в кооперация с операционната система. При компилируемите езици за програмиране **компилирането на програмата** се извършва задължително преди нейното изпълнение и по време на компилация се откриват синтактичните грешки (грешно зададени команди). С компилатор работят езици като C++, C#, Java, Swift и Go.

Някои езици за програмиране не използват компилатор, а се **интерпретират директно** от специализиран софтуер, наречен "интерпретатор". **Интерпретаторът** е "програма за изпълняване на програми", написани на някакъв програмен език. Той изпълнява командите на програмата една след друга, като разбира не само от единични команди и поредици от команди, но и от другите езикови конструкции (проверки, повторения, функции и т.н.). Езици като PHP, Python и JavaScript работят с интерпретатор и се изпълняват без да се компилират. Поради липса на предварителна компилация, при интерпретираните езици **грешките се откриват по време на изпълнение**, след като програмата започне да работи, а не предварително.

Средата за програмиране (Integrated Development Environment – IDE, интегрирана среда за разработка) е съвкупност от традиционни инструменти за разработване на софтуерни приложения. В средата за разработка пишем код, компилираме и изпълняваме програмите. Средите за разработка интегрират в себе си **текстов редактор** за писане на кода, **език за програмиране**, **компилатор** или **интерпретатор** и **среда за изпълнение** за изпълнение на програмите, **дебъгер** за проследяване на програмата и търсене на грешки, **инструменти за дизайн на потребителски интерфейс** и други инструменти и добавки.

Средите за програмиране са удобни, защото интегрират всичко необходимо за разработката на програмата, без да се напуска средата. Ако не ползваме среда за разработка, ще трябва да пишем кода в текстов редактор, да го компилираме с

команда от конзолата, да го изпълняваме с друга команда от конзолата и да пишем още допълнителни команди, когато се налага, и това ще ни губи време. Затова повечето програмисти ползват IDE в ежедневната си работа.

За програмиране на **езика C#** най-често се ползва средата за разработка **Visual Studio**, която се разработва и разпространява безплатно от Microsoft и може да се изтегли от: <https://www.visualstudio.com/downloads>. Алтернативи на Visual Studio са **Rider** (<https://jetbrains.com/rider>) и **MonoDevelop / Xamarin Studio** (<http://www.monodevelop.com>) и **SharpDevelop** (<http://www.icsharpcode.net/OpenSource/SD>). В настоящата книга ще използваме средата за разработка Visual Studio.

Езици от ниско и високо ниво, среди за изпълнение

Програмата в своята същност е **набор от инструкции**, които карат компютъра да свърши определена задача. Те се въвеждат от програмиста и се **изпълняват бешевло от машината**.

Съществуват различни видове **езици за програмиране**. С езиците от най-ниско ниво могат да бъдат написани **самите инструкции**, които **управляват процесора**, например с езика "**assembler**". С езици от малко по-високо ниво като **C** и **C++** могат да бъдат създадени операционна система, драйвери за управление на хардуера (например драйвер за видеокарта), уеб браузъри, компилатори, двигатели за графика и игри (game engines) и други системни компоненти и програми. С езици от още по-високо ниво като **C#, Python** и **JavaScript** се създават приложни програми, например програма за четене на поща или чат програма.

Езиците от ниско ниво управляват директно хардуера и изискват много усилия и огромен брой команди, за да свършат единица работа. **Езиците от по-високо ниво** изискват по-малко код за единица работа, но нямат директен достъп до хардуера. На тях се разработва приложен софтуер, например уеб приложения и мобилни приложения.

Болшинството софтуер, който използваме ежедневно, като музикален плеър, видеоплеър, GPS програма и т.н., се пише на **езици за приложно програмиране**, които са от високо ниво, като **C#, Java, Python, C++, JavaScript, PHP** и др.

C# е компилируем език, а това означава, че пишем команди, които се компилират преди да се изпълнят. Именно тези команди, чрез помощна програма (компилатор), се преобразуват във файл, който може да се изпълнява (executable). За да пишем на език като **C#** ни трябва текстов редактор или среда за разработка и **.NET среда за изпълнение**.

.NET средата за изпълнение (.NET Runtime Environment) представлява виртуална машина, нещо като компютър в компютъра, която може да изпълнява компилиран **C#** код. С риск да навлезем в твърде много детайли, трябва да поясним, че езикът **C#** се компилира до междинен .NET код и се изпълнява от .NET средата, която компилира този междинен код допълнително в движение до машинни инструкции (машинен код) за да се изпълни от микропроцесора. **.NET** средата съдържа библиотеки с класове, **CSC** компилатор, **CLR** (Common Language Runtime) и други

компоненти, които са необходими, за да работим с езика C# и изпълняваме C# програми.

Средата .NET е достъпна като свободен софтуер с отворен код за всички съвременни операционни системи (като Windows, Linux и Mac OS X). Тя има две разновидности, .NET Framework (по-старата) и .NET Core (по-новата), но всичко това няма съществено значение за навлизането в програмирането. Нека се фокусираме върху писането на програми с езика C#.

Компютърни програми – компилация и изпълнение

Както вече споменахме, програмата е **последователност от команди**, иначе казано тя описва поредица от пресмятания, проверки, повторения и всякакви подобни операции, които целят постигане на някакъв резултат.

Програмата се пише в текстов формат, а самият текст на програмата се нарича **сурс код** (source code). Той се компилира до **изпълним файл** (например **Program.cs** се компилира до **Program.exe**) или се **изпълнява директно** от .NET средата.

Процесът на **компилация** на кода преди изпълнение се използва само при компилируеми езици като C#, Java и C++. При **скриптови и интерпретериеми езици**, като JavaScript, Python и PHP, кодът се изпълнява постъпково от интерпретатор.

Компютърни програми – примери

Да започнем с много прост пример за кратка C# програма.

Пример: програма, която свири музикалната нота "ла"

Нашата първа програма ще е единична C# команда, която свири музикалната нота "ла" (432 херца) с продължителност половин секунда (500 милисекунди):

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

След малко ще разберем как можем да изпълним тази команда и да чуем звука от нотата, но засега нека само разгледаме какво представляват командите в програмирането. Да се запознаем с още няколко примера.

Пример: програма, която свири поредица от музикални ноти

Можем да усложним предходната програма, като зададем за изпълнение повтарящи се в цикъл команди за свирене на поредица от ноти с нарастваща височина:

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

В горния пример караме компютъра да свири една след друга за много кратко (по 100 милисекунди) всички ноти с височина 200, 400, 600 и т.н. херца до достигане на 4000 херца. Резултатът от програмата е свирене на нещо като мелодия.

Как работят повторенията (циклите) в програмирането ще научим в [главата "Цикли"](#), но засега приемете, че просто повтаряме някаква команда много пъти.

Пример: програма, която конвертира от левове в евро

Да разгледаме още една проста програма, която прочита от потребителя някаква сума в лева (цяло число), конвертира я в евро (като я разделя на курса на еврото) и отпечатва получения резултат. Това е програма от 3 поредни команди:

```
var leva = int.Parse(Console.ReadLine());
var euro = leva / 1.95583;
Console.WriteLine(euro);
```

Разгледахме [три примера за компютърни програми](#): единична команда, серия команди в цикъл и поредица от 3 команди. Нека сега преминем към по-интересното: как можем да пишем собствени програми на **C#** и как можем да ги компилираме и изпълняваме?

Как да напишем конзолна програма?

Нека преминем през [стъпките за създаване и изпълнение на компютърна програма](#), която чете и пише своите данни от и на текстова конзола (прозорец за въвеждане и извеждане на текст). Такива програми се наричат "[конзолни](#)". Преди това, обаче, трябва първо да си [инсталдраме и подгответим средата за разработка](#), в която ще пишем и изпълняваме C# програмите от тази книга и упражненията към нея.

Среда за разработка (IDE)

Както вече стана дума, за да програмираме ни е нужна [среда за разработка](#) – **Integrated Development Environment** (IDE). Това е всъщност редактор за програми, в който пишем програмния код и можем да го компилираме и изпълняваме, да виждаме грешките, да ги поправяме и да стартираме програмата отново.

- За програмиране на C# използваме средата **Visual Studio** за операционната система Windows и **MonoDevelop** или **Rider** за Linux или Mac OS X.
- Ако програмираме на Java, подходящи са средите **IntelliJ IDEA**, **Eclipse** или **NetBeans**.
- Ако ще пишем на Python, можем да използваме средата **PyCharm**.

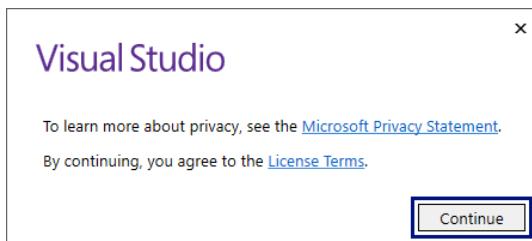
Инсталация на Visual Studio Community

Започваме с инсталацията на интегрираната среда Microsoft Visual Studio Community (версия 2017, актуална към юни 2017 г.).

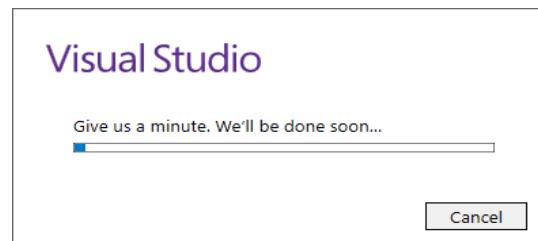
Community версията на Visual Studio (VS) се разпространява бесплатно от Microsoft и може да бъде изтеглена от: <https://www.visualstudio.com/vs/community>. Инсталацията е типичната за Windows с [Next], [Next] и [Finish], но е важно да включим компонентите за "desktop development" и "ASP.NET". Не е необходимо да променяме останалите настройки за инсталация.

В следващите редове подробно са описани подробно **стъпките за инсталация на Visual Studio** (версия Community 2017).

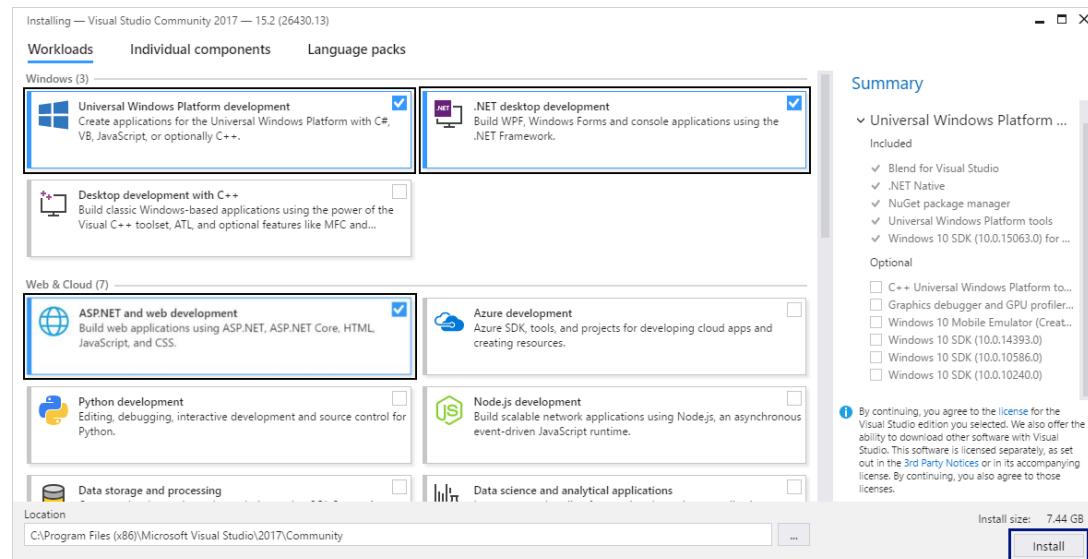
След като свалим и стартираме инсталационния файл, се появява този еcran:



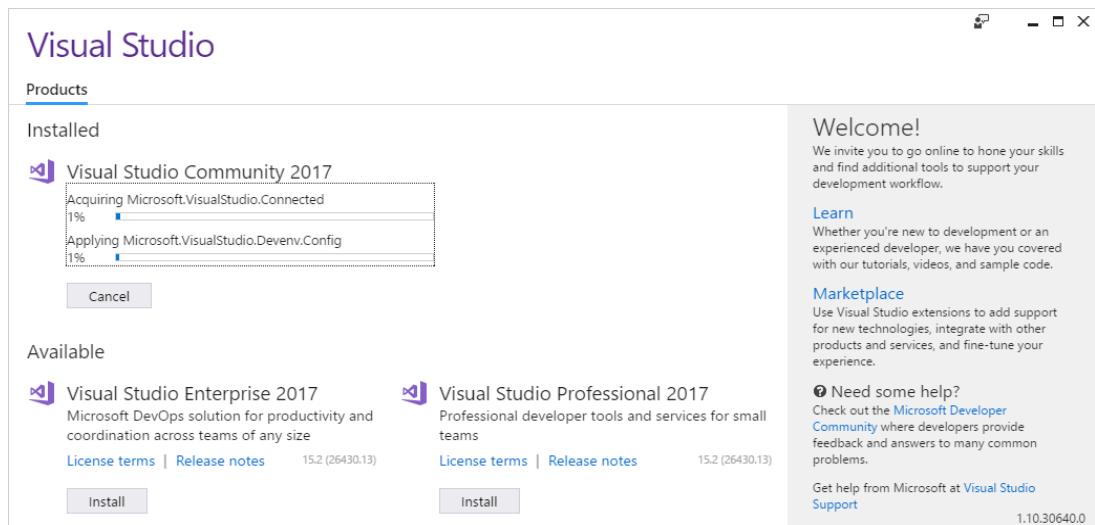
Натискаме бутона [Continue] и ще се появи прозорецът по-долу:



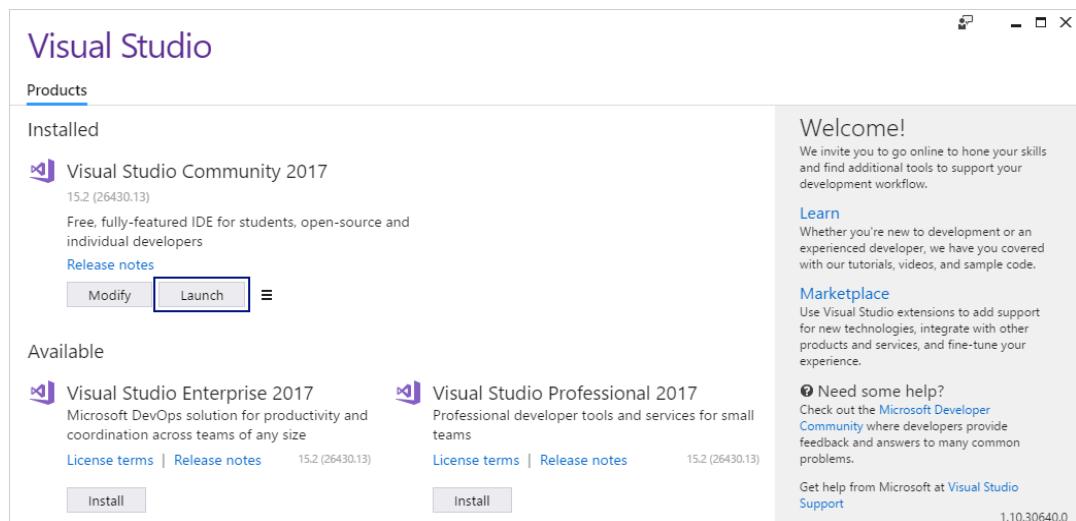
Зарежда се прозорец с инсталационния панел на Visual Studio. Слагаме отметка на [.NET desktop development], [ASP.NET and web development] и на [Universal Windows Platform development], след което натискаме бутона [Install]. Общо взето това е ВСИЧКО.



Започва **инсталацията на Visual Studio** и се появява еcran като този по-долу:

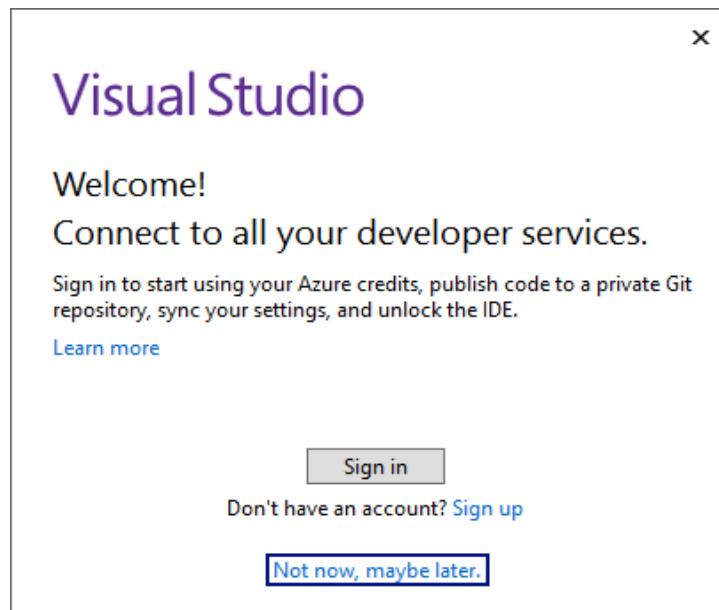


След като Visual Studio се инсталира, ще се появи информативен еcran и трябва да натиснем бутона [Launch], за да го стартираме.

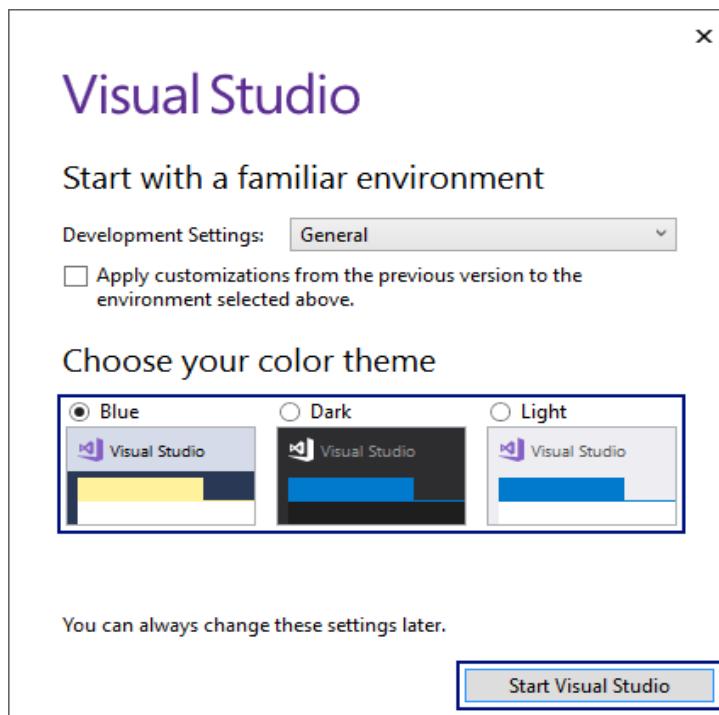


След **старта на VS** излиза еcran като този по-долу. От него можем да изберем дали да влезем с Microsoft профила си във Visual Studio. За момента избираме да работим без да сме се логнали с Microsoft акаунта си, затова избираме опцията [**Not now, maybe later.**].

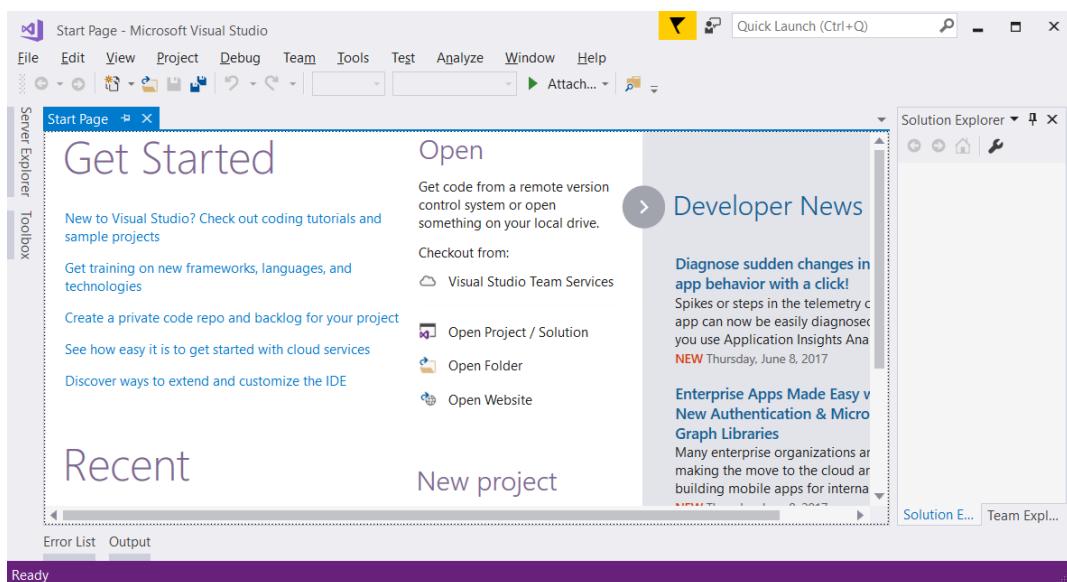
На по-късен етап, ако имате такъв акаунт, можете да се логнете, а ако нямате и срещате затруднения със създаването му, винаги можете да пишете във форума на SoftUni: <https://softuni.bg/forum>.



Следващата стъпка е да изберем **цветовата тема**, с която да се визуализира Visual Studio. Тук изборът е изцяло според предпочтенията на потребителя, като няма значение коя опция ще бъде избрана.



Натискаме бутона [Start Visual Studio] и се зарежда в началния изглед на Visual Studio Community:



Това е всичко. Готови сме за работа с Visual Studio.

По-стари версии на Visual Studio

Можем да използваме и по-стари версии на Visual Studio (например версия 2015 или 2013 или дори 2010 или 2005), но **не е препоръчително**, тъй като в тях не се съдържат някои от по-новите възможности за разработка и не всички примери от книгата ще тръгнат.

Онлайн среди за разработка

Съществуват и алтернативни среди за разработка онлайн, директно във вашия уеб браузър. Тези среди не са много удобни, но ако нямате друга възможност, може да стартирате обучението си с тях и да си качите Visual Studio по-късно. Ето някои линкове:

- За езика **C#** сайтът **.NET Fiddle** позволява писане на код и изпълнението му онлайн: <https://dotnetfiddle.net>.
- За **Java** можем да използваме онлайн Java IDE: <https://www.compilejava.net>.
- За **JavaScript** можем да пишем JS код директно в конзолата на даден браузър с натискане на **[F12]**.

Проектни решения и проекти във Visual Studio

Преди да започнем да работим с Visual Studio е нужно да се запознаем с понятията **Visual Studio Solution** и **Visual Studio Project**, които са неизменна част от него.

Visual Studio Project представлява "проектът", върху който работим. В началото това ще са нашите конзолни програми, които ще се научим да пишем с помощта на настоящата книга, ресурсите към нея и в курса Programming Basics в SoftUni.

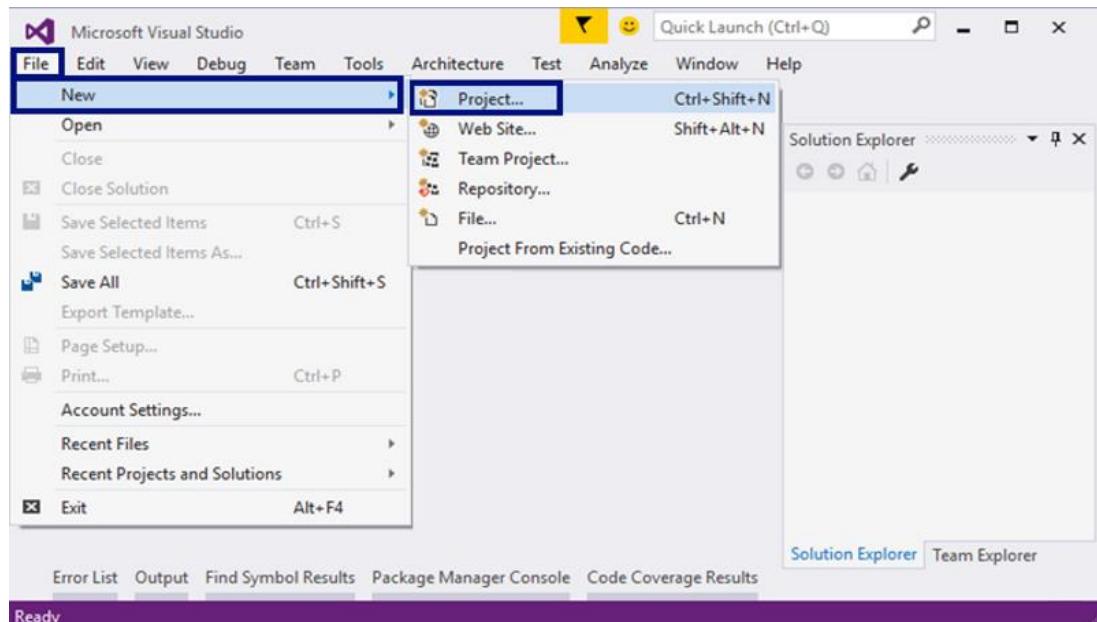
При по-задълбочено изучаване и с времето и практиката, тези проекти ще преминат в апликации, уеб приложения и други разработки. Проектът във VS логически групира множество файлове, изграждащи дадено приложение или компонент. Един C# проект съдържа един или няколко C# сурс файла, конфигурационни файлове и други ресурси. Във всеки C# сурс файл има една или повече **дефиниции на типове** (класове или други дефиниции). В **класовете** има **методи** (действия), а те се състоят от **поредици от команди**. Изглежда сложно, но при големи проекти такава структура е много удобна и позволява добра организация на работните файлове.

Visual Studio Solution представлява контейнер (работно решение), в който логически са обединени няколко проекта. Целта на обединението на тези VS Projects е да има възможност кода от който и да е от проектите, да си взаимодейства с кода на останалите VS проекти, за да може приложението или уеб сайта да работи коректно. Когато софтуерният продукт или услуга, който разработваме е голям, той се изгражда като **VS Solution**, а този Solution се разделя на **проекти** (VS Projects) и във всеки проект има **папки със сурс файлове**. Такава йерархична организация е удобна при по-сериозни проекти (да кажем над 50 000 реда код).

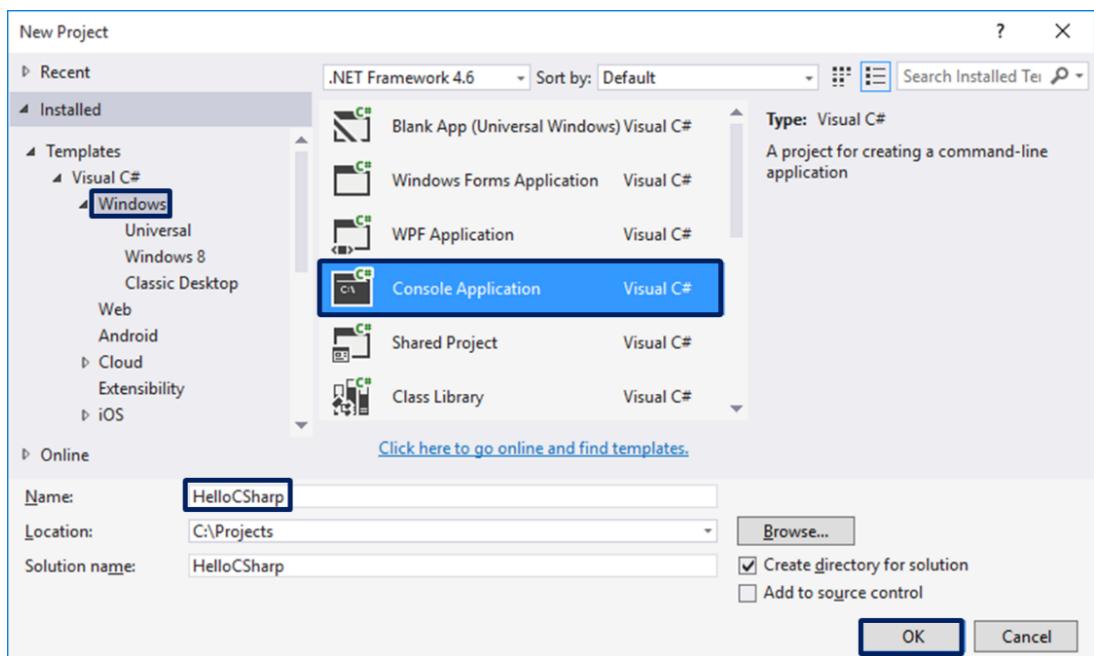
За **малки проекти** VS Solutions и VS Projects повече усложняват работата, отколкото помагат, но се свиква бързо.

Пример: създаване на конзолна програма "Hello C#"

Да се върнем на нашата конзолна програма. Вече имаме Visual Studio и можем да го стартираме. След това създаваме нов конзолен проект: [File] → [New] → [Project] → [Visual C#] → [Windows] → [Console Application].



Задаваме **смислено име** на нашата програма, например **HelloCSharp**:



Visual Studio ще създаде за нас **празна C# програма**, която трябва да допищем (VS Solution с VS Project в него със C# сурс файл в него с един C# клас в него с **Main()** метод в него).

Писане на програмен код

Сурс кодът на C# програмите се пише в секцията **Main(string[] args)**, между отварящата и затварящата скоба **{ }**. Това е главният метод (действие), което се изпълнява при стартиране на една C# програма. Този главен **Main()** метод може да се запише по два начина:

- **static void Main(string[] args)** – с параметри от командния ред (няма да навлизаме в подробности)
- **static void Main()** – без параметри от командния ред

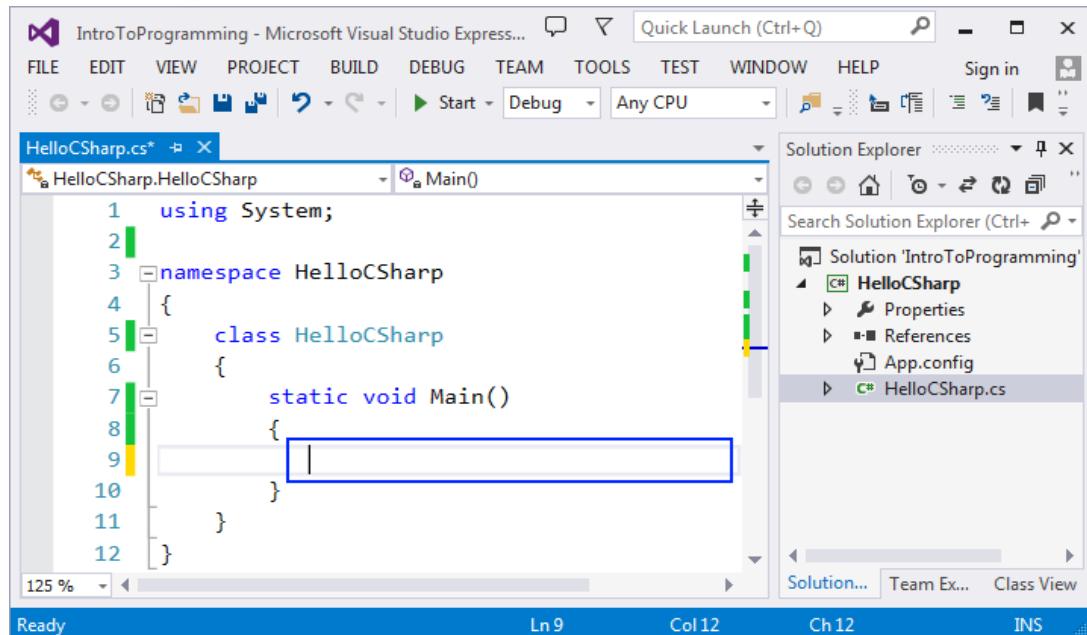
И двета начина са валидни, като **вторият е за предпочитане**, защото е по-кратък и по-изчистен. По подразбиране, обаче, при създаване на конзолна програма Visual Studio ползва първия начин, който можем по желание да редактираме на ръка и да изтрием частта с параметрите **string[] args**.

Натискаме **[Enter]** след отварящата скоба **{** и започваме да пишем. Кодът на програмата се пише **отместен навътре**, като това е част от оформянето на текста, за по-голямо удобство при повторен преглед и/или дебъгване.

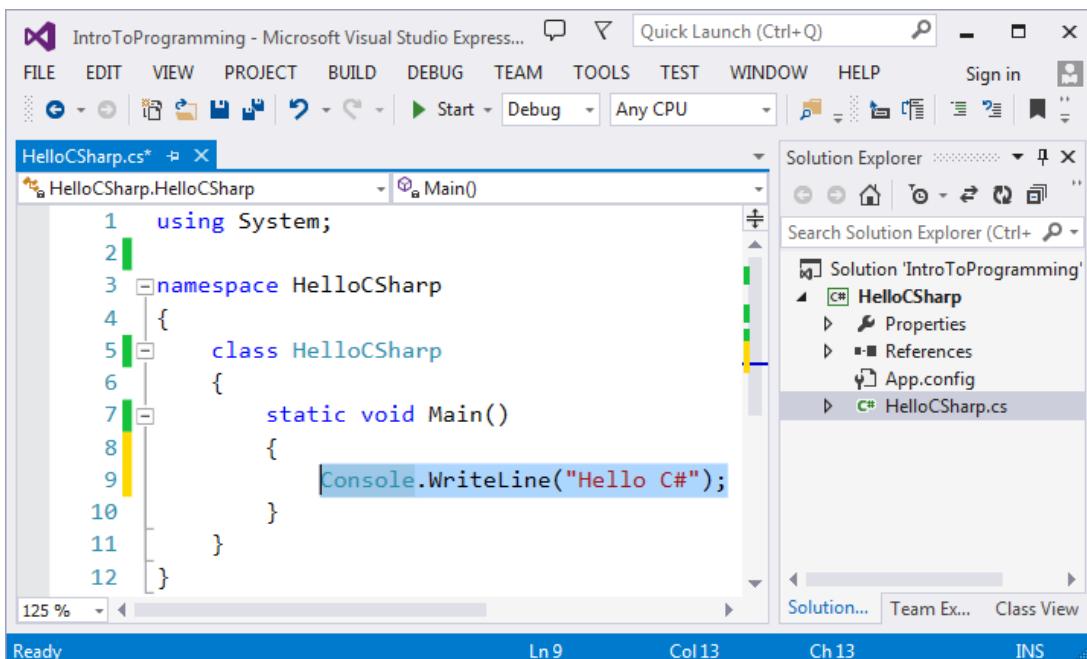
Пишем следната команда:

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

Ето как изглежда Visual Studio преди на напишем код на нашата програма:



Ето как трябва да изглежда нашата програма във Visual Studio:

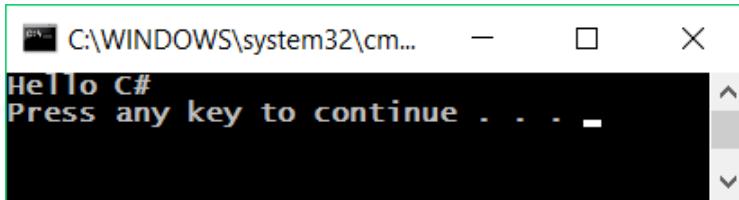


Командата `Console.WriteLine("Hello C#")` на езика C# означава да изпълним отпечатване (`WriteLine(...)`) върху конзолата (`Console`) и да отпечатаме текстово съобщение `Hello C#`, което трябва да оградим с кавички, за да поясним, че това е текст. В края на всяка команда на езика C# се слага символът `;` и той указва, че команда свършва на това място (т.е. не продължава на следващия ред).

Тази команда много типична за програмирането: указваме да се намери даден **обект** (в случая конзолата) и върху него да се изпълни някакво **действие** (в случая печатане на нещо, което се задава в скоби). По-техническо обяснено, извикваме метода **WriteLine(...)** от класа **Console** и му подаваме като параметър текстов лiteral "Hello C#".

Стартиране на програмата

За стартиране на програмата натискаме [Ctrl + F5]. Ако няма грешки, програмата ще се изпълни. Резултатът ще се изпише на конзолата (в черния прозорец)



Забележете, че стартираме с [Ctrl+F5], а не само с [F5] или с бутона за стартиране във Visual Studio. Ако ползваме [F5], програмата ще се изпълни за кратко и веднага след това черният прозорец ще изчезне и няма да видим резултата.

Въсъщност, **изходът от програмата** е следното текстово съобщение:

```
Hello C#
```

Съобщението "Press any key to continue..." се изписва допълнително на най-долния ред на конзолата от Visual Studio след като програмата завърши, за да ни подканят да видим резултата от изпълнението на програмата и да натиснем клавиш, за да затворим конзолата.

Тестване на програмата в Judge системата

Тестването на задачите от тази книга е автоматизирано и се осъществява през Интернет, от сайта на **Judge системата**: <https://judge.softuni.bg>. Оценяването на задачите се извършва на момента от системата. Всяка задача минава поредица от тестове, като всеки успешно преминат тест дава предвидените за него точки. Тестовете, които се подават на задачите, са скрити.

Горната програма може да тестваме тук: <https://judge.softuni.bg/Contests/Practice/Index/503#0>. Поставяме целия сурс код на програмата в черното поле и избираме **C# code**, както е показано по-долу.

Изпращаме решението за оценяване с бутона [**Изпрати**]. Системата връща резултат след няколко секунди в таблицата с изпратени решения.

01. Hello C#

```

1  using System;
2
3  namespace HelloCSharp
4  {
5      class HelloCSharp
6      {
7          static void Main()
8          {
9              Console.WriteLine("Hello C#");
10         }
11     }
12 }
13

```

Позволено време: 0.100 sec.
Позволена памет: 16.00 MB
Size limit: 16.00 KB
Checker: Trim

C# code Изпрати

| Изпратени решения | | |
|--|----------------------------------|--|
| Точки | Използвано време и памет | Изпратено на |
| ✓ 100 / 100 | Памет: 7.20 MB Време: 0.015 s | 12:40:04 05.06.2017 Детайли |

При необходимост може да натиснем бутона за обновяване на резултатите [Refresh] в горната дясна част на таблицата с изпратени за проверка решения:

| Изпратени решения | | |
|--|----------------------------------|--|
| Точки | Използвано време и памет | Изпратено на |
| ✓ 100 / 100 | Памет: 7.18 MB Време: 0.015 s | 12:40:04 05.06.2017 Детайли |
| ✗ 0 / 100 | Памет: 7.22 MB Време: 0.015 s | 13:13:30 05.06.2017 Детайли |

В таблицата с изпратените решения judge системата ще покаже един от следните възможни резултати:

- Брой точки (между 0 и 100), когато предаденият код се компилира успешно (няма синтактични грешки) и може да бъде тестван.
 - При **вярно решение** всички тестове са маркирани в зелено и получаваме 100 точки.

- При **грешно решение** някои от тестовете са маркирани в червено и получаваме непълен брой точки или 0 точки.
- При грешна програма ще получим **съобщение за грешка** по време на компилиация.

Как да се регистрирам в SoftUni Judge?

Използваме идентификацията си (username + password) за сайта softuni.bg. Ако нямаете СофтУни регистрация, направете си. Отнема само минутка – стандартна регистрация в Интернет сайт.

Тествайте програмите за свирене на ноти

Сега, след като вече **знаете как да изпълнявате програми**, можете да тествате примерните програми по-горе, които свирят музикални ноти. Позабавлявайте се, пробвайте тези програми. Пробвайте да ги промените и да си поиграете с тях. Заменете командата **Console.WriteLine("Hello C#");** с **Console.Beep(432, 500);** и стартирайте програмата. Проверете дали ви е включен звука на компютъра и дали е усилен. Ако работите в онлайн среда за разработка, няма да чуете звук, защото програмата не се изпълнява на вашия компютър, а накъде другаде.

Типични грешки в C# програмите

Една от често срещаните грешки при начинаещите е **писането извън тялото на Main() метода**, защото интегрираната среда или компилаторът не биха могли правилно да разчетат зададените команди в програмата. Ето пример за грешно написана програма:

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

Друга грешка е бъркането на **главни и малки букви**, а те имат значение при извикване на командите и тяхното правилно функциониране. Ето пример за такава грешка:

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

В горния пример **Writeline** е изписано грешно и трябва да се поправи на **WriteLine**.

Липсата на точка и запетая (`;`) в края на командите е един от вечните проблеми на начинаещия програмист. Пропускането на този знак води до **неправилно функциониране на програмата** и често проблемът остава незабелязан. Ето примерен грешен код:

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

Липсваща кавичка или липса на отваряща или затваряща скоба също може да се окажат проблеми. Както и при точката и запетаята, така и тук проблемът води до **неправилно функциониране на програмата** или въобще до нейното неизпълнение. Този пропуск трудно се забелязва при по-обемен код. Ето пример за грешна програма:

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

Тази програма ще даде **грешка** при опит за **компиляция** и стартиране и даже още преди това кодът ще бъде подчертан, за да се насочи вниманието на програмиста към грешката, която е допуснал (пропуснатата затваряща кавичка):

| # | Code | Description | Project | File | Line |
|---|--------|---------------------|-------------|------------|------|
| 1 | CS1010 | Newline in constant | HelloCSharp | Program.cs | 14 |
| 2 | CS1026 |) expected | HelloCSharp | Program.cs | 14 |
| 3 | CS1002 | ; expected | HelloCSharp | Program.cs | 14 |

Какво научихме от тази глава?

На първо място научихме **какво е програмирането** – задаване на **команди**, изписани на **компютърен език**, които машината разбира и може да изпълни. Разбрахме още какво е **компютърната програма** – тя представлява **поредица от команди**, подредени една след друга.

Запознахме се с **езика за програмиране C#** на базисно ниво и как да създаваме **прости конзолни програми** с Visual Studio. Проследихме и **структурата на програмния код в езика C#**, като например, че командите главно се задават в секцията **static void Main(string[] args)** между **отварящата и затварящата къдрава скоба**. Видяхме как да печатаме с **Console.WriteLine(...)** и как да стартираме програмата си с [Ctrl + F5]. Научихме се и как да тестваме кода си в **SoftUni Judge**.

Добра работа! Да се захващаме с **упражненията**. Нали не сте забравили, че програмиране се учи с много писане на код и решаване на задачи? Да решим няколко задачи, за да затвърдим наученото.

Упражнения: първи стъпки в коденето

Добре дошли в **упражненията**. Сега ще напишем **няколко конзолни програми**, с които ще направим още няколко първи стъпки в програмирането, след което ще покажем как можем да програмираме нещо по-сложно – програми с графичен и учеб потребителски интерфейс.

Задача: конзолна програма “Expression”

Да се напише конзолна C# програма, която **пресмята** и отпечатва стойността на следния числен израз:

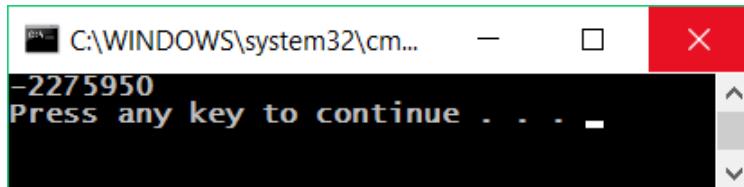
$$(3522 + 52353) * 23 - (2336 * 501 + 23432 - 6743) * 3$$

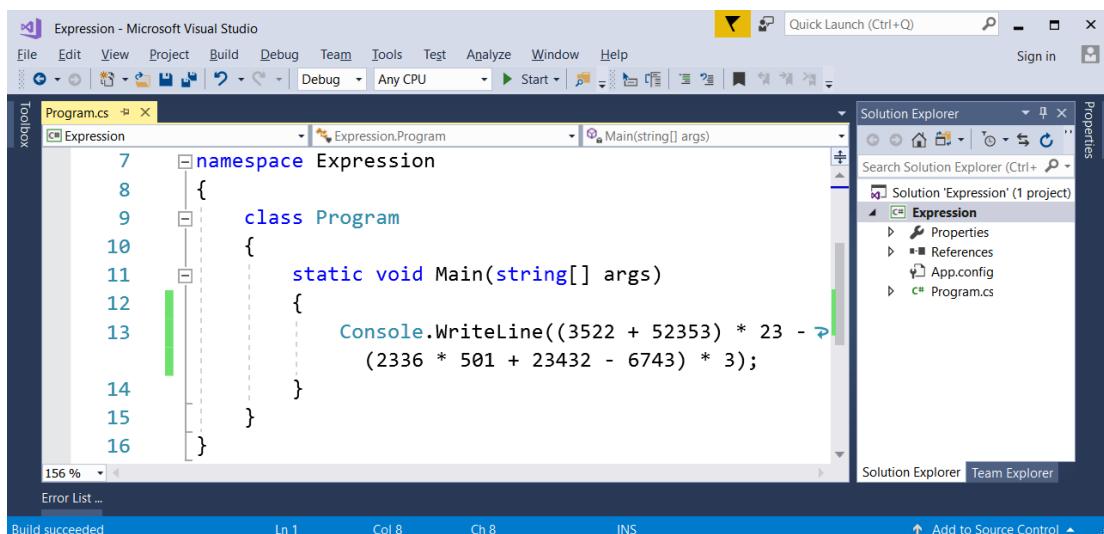
Забележка: **не е разрешено да се пресметне стойността предварително** (например с Windows Calculator).

Насоки и подсказки

Правим нов C# конзолен проект с име "Expression". Намираме метода **static void Main(string[] args)** и влизаме в неговото тяло между **{** и **}**. След това трябва да **напишем кода**, който да изчисли горния числен израз и да отпечата на конзолата стойността му. Подаваме горния числен израз в скобите на командата **Console.WriteLine(...)**, както е показано на картинката по-долу.

Стартираме програмата с [Ctrl+F5] и проверяваме е верен резултатът:





Тестване в Judge системата

Тествайте решението си: <https://judge.softuni.bg/Contests/Practice/Index/503#1>.

Поставяте целия сорс код на вашата програма в полето за изпращане на код за втората задача „Expression“ и натискате бутона за изпращане. Ето как би могло да изглежда това във вашия браузър (в зависимост от настройките на браузъра judge се показва на български или на английски език):

Secure | <https://judge.softuni.bg/Contests/Practice/Index/503#1>

02. Expression

```

1 using System;
2 namespace Expression
3 {
4     class Program
5     {
6         static void Main(string[] args)
7         {
8             Console.WriteLine((3522 + 52353) * 23 - (2336*501 + 23432 - 6743) * 3);
9         }
10    }
11 }
12

```

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

C# code

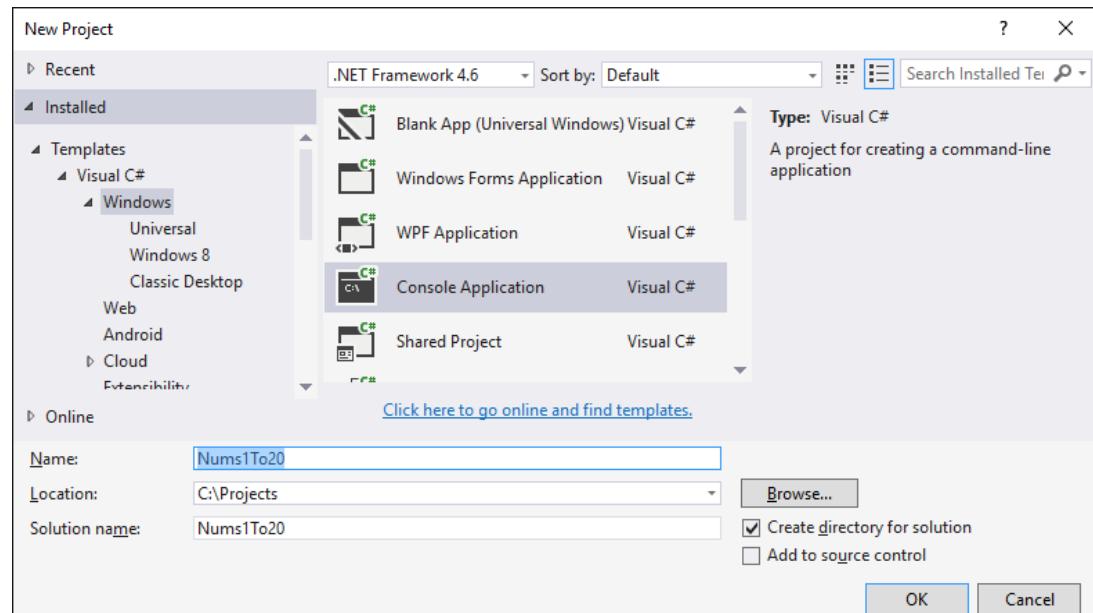
| Submissions | | |
|-------------|----------------------------------|--|
| 1 | | |
| Points | Time and memory used | Submission date |
| ✓ 100 / 100 | Memory: 7.22 MB Time: 0.015 s | 14:04:00 06.06.2017 <input type="button" value="Details"/> |
| 1 | | |

Задача: числата от 1 до 20

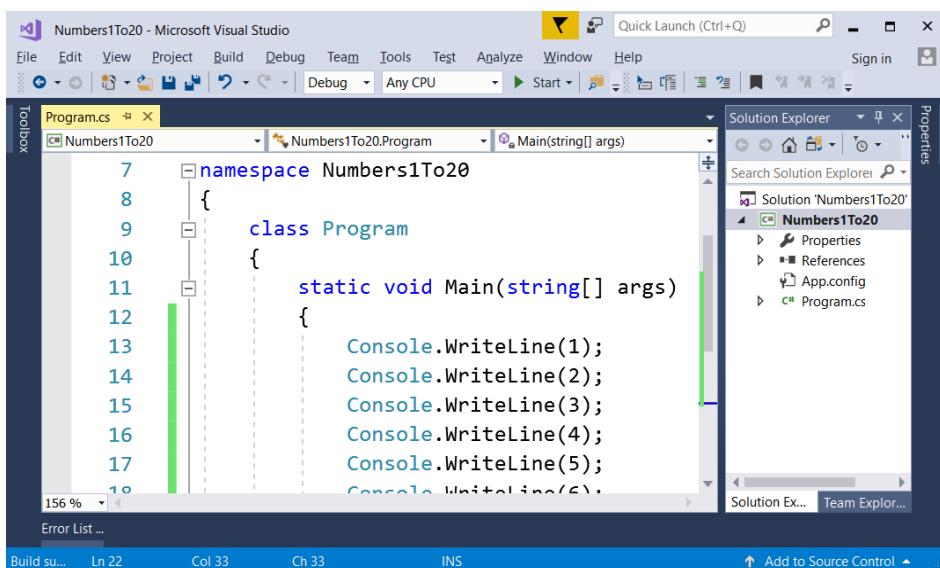
Да се напише C# конзолна програма, която отпечатва числата от 1 до 20 на отделни редове на конзолата.

Насоки и подсказки

Създаваме конзолно C# приложение с име "Nums1To20":



В **static void Main()** метода пишем 20 команди **Console.WriteLine(...)**, всяка на отделен ред, за да отпечатаме числата от 1 до 20 едно след друго. Подсветливите от вас, сигурно се питат дали няма по-умен начин. Спокойно, има, но за него по-късно.



Сега **стартираме програмата** и поверяваме дали резултатът е какъвто се очаква да бъде:

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

Тестване в Judge системата

Тествайте решението си: <https://judge.softuni.bg/Contests/Practice/Index/503#2>

Сега помислете дали може да напишем програмата по-**умен начин**, така че да не повтаряме 20 пъти една и съща команда. Потърсете в Интернет информация за "[for loop C#](#)".

Задача: Триъгълник от 55 звездички

Да се напише C# конзолна програма, която **отпечатва триъгълник от 55 звездички**, разположени на 10 реда:

```
*  
**  
***  
****  
*****  
*****  
*****  
*****  
*****  
*****
```

Насоки и подсказки

Създаваме ново конзолно C# приложение с име "TriangleOf55Stars". В него трябва да напишем код, който печата триъгълника от звездички, например чрез 10 команди, като посочените по-долу:

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

Тестване в Judge системата

Тествайте решението си: <https://judge.softuni.bg/Contests/Practice/Index/503#3>.

Опитайте да подобрите решението, така че да няма много повтарящи се команди. Може ли това да стане с **for** цикъл? Успяхте ли да намерите умно решение (например с цикъл) на предната задача? При тази задача може да се ползва нещо подобно, но малко по-сложно (два цикъла един в друг).

Задача: лице на правоъгълник

Да се напише C# програма, която прочита от конзолата две числа **a** и **b**, пресмята и отпечатва лицето на правоъгълник със страни **a** и **b**.

Примерен вход и изход

| a | b | area |
|---|---|------|
| 2 | 7 | 14 |

| a | b | area |
|----|---|------|
| 12 | 5 | 60 |

| a | b | area |
|---|---|------|
| 7 | 8 | 56 |

Насоки и подсказки

Правим нова конзолна C# програма. За да прочетем входните числа, използваме следните две команди:

```
static void Main(string[] args)
{
    var a = decimal.Parse(Console.ReadLine());
    var b = decimal.Parse(Console.ReadLine());

    //TODO: Пресметнете лицето и го принтирайте в конзолата
}
```

Остава да се допише програмата по-горе, за да пресмята лицето на правоъгълника и да го отпечатва. Използвайте познатата ни вече команда **Console.WriteLine()** и ѝ подайте в скобите произведението на числата **a** и **b**. В програмирането умножението се извършва с оператора *****.

Тествайте решението си

Тествайте решението си с няколко примера. Трябва да получите резултат, подобен на този (въвеждаме 2 и 7 като вход и програмата отпечатва като резултат 14 – тяхното произведение):

```
2
7
14
```

Тестване в Judge системата

Тествайте решението си: <https://judge.softuni.bg/Contests/Practice/Index/503#4>.

* Задача: квадрат от звездички

Да се напише C# конзолна програма, която прочита от конзолата **цяло положително число n** и отпечатва на конзолата **квадрат от n звездички**, като в примерите по-долу.

Примерен вход и изход

| Вход | Изход |
|------|-------------------|
| 3 | *** * * *** |

| Вход | Изход |
|------|----------------------------|
| 4 | **** * * * * **** |

| Вход | Изход |
|------|-------------------------------------|
| 5 | ***** * * * * * * ***** |

Насоки и подсказки

Правим нова конзолна C# програма. За да прочетем числото **n** ($2 \leq n \leq 100$), използваме следния код:

```
static void Main(string[] args)
{
    var n = int.Parse(Console.ReadLine());

    //TODO: Принтирайте правоъгълника
}
```

Да се допише програмата по-горе, за да отпечатва квадрат, съставен от звездички. Може да се наложи да се използват **for** цикли. Потърсете информация в Интернет.

Внимание: тази задача е по-трудна от останалите и нарочно е дадена сега и е обозначена със звездичка, за да ви провокира да потърсите информация в Интернет. Това е едно от най-важните умения, което трябва да развивате докато учите програмирането: [да търсите информация в Интернет](#). Това ще правите всеки ден, ако работите като програмисти, така че не се плашете, а се опитайте. Ако имате трудности, можете да потърсите помощ и в СофтУни форума: <https://softuni.bg/forum>.

Тестване в Judge системата

Тествайте решението си: <https://judge.softuni.bg/Contests/Practice/Index/503#5>.

Конзолни, графични и уеб приложения

При конзолните приложения (Console Applications), **всички операции** за четене на вход и печтане на изход се извършват през конзолата. Там се **въвеждат** входните

данни, които се прочитат от приложението, там се **отпечатват и изходните данни** след или по време на изпълнение на програмата.

Докато конзолните приложения **ползват текстовата конзола**, уеб приложениета (Web Applications) **използват уеб-базиран потребителски интерфейс**. За да се **постигне тяхното изпълнение** са необходими две неща – **уеб сървър** и **уеб браузър**, като **браузърът** играе главната роля по **визуализация на данните и взаимодействието с потребителя**. Уеб приложениета са много по-приятни за потребителя, изглеждат визуално много по-добре, използват се мишка и докосване с пръст (при таблети и телефони), но зад всичко това стои програмирането. И затова **трябва да се научим да програмираме** и вече направихме първите си съвсем малки стъпки.

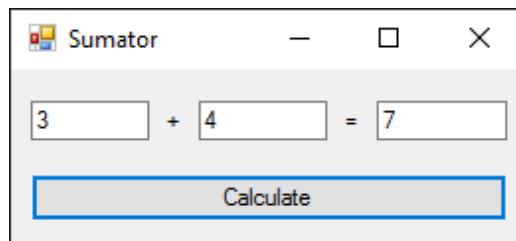
Графичните (GUI) приложения имат **визуален потребителски интерфейс**, директно върху вашия компютър или мобилно устройство, без да е необходим уеб браузър. Графичните приложения (настолни приложения или, иначе казано, desktop apps) се състоят от **един или повече графични прозореца**, в които се намират определени **контроли** (текстови полета, бутони, картички, таблици и други), **служещи за диалог** с потребителя по-интуитивен начин. Подобни са и мобилните приложения във вашия телефон и таблет: ползваме форми, текстови полета, бутони и други контроли и ги управляване чрез програмен код. Нали затова се учим сега да пишем код: **кодът е навсякъде в разработката на софтуер**.

Упражнения: графични и уеб приложения

Сега предстои да направим едно просто уеб приложение и едно просто **графично приложение**, за да можем да надникнем в това, какво ще можем да създаваме като напреднем с програмирането и разработката на софтуер. Няма да разглеждаме детайлите по използваните техники и конструкции из основи, а само ще хвърлим поглед върху подредбата и функционалността на създаденото от нас. След като напреднем със знанията си, ще бъдем способни да правим големи и сложни софтуерни приложения и системи. Надяваме се примерите по-долу **да ви запалят интереса**, а не да ви откажат.

Задача: графично приложение „Суматор за числа“

Да се напише **графично (GUI)** приложение, което **изчислява сумата на две числа**.



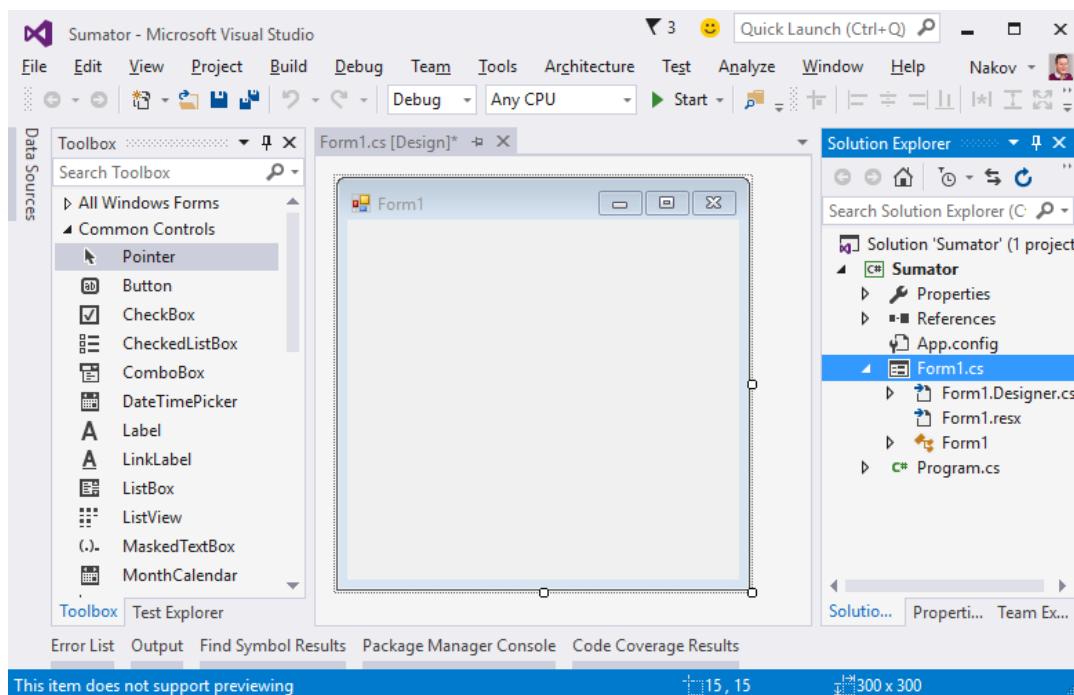
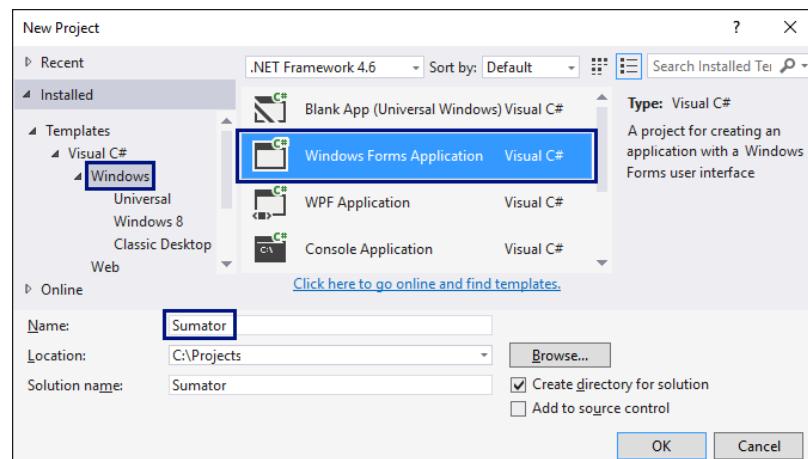
При въвеждане на две числа в първите две текстови полета и натискане на бутона [Calculate] се изчислява тяхната сума и резултатът се показва в третото текстово поле. При грешно въведени числа, се изписва грешка в полето за резултата.

За нашето приложение ще използваме **технологията Windows Forms**, която позволява създаване на **графични приложения за Windows** (GUI application), в среда за разработка **Visual Studio** и с език за програмиране **C#**.

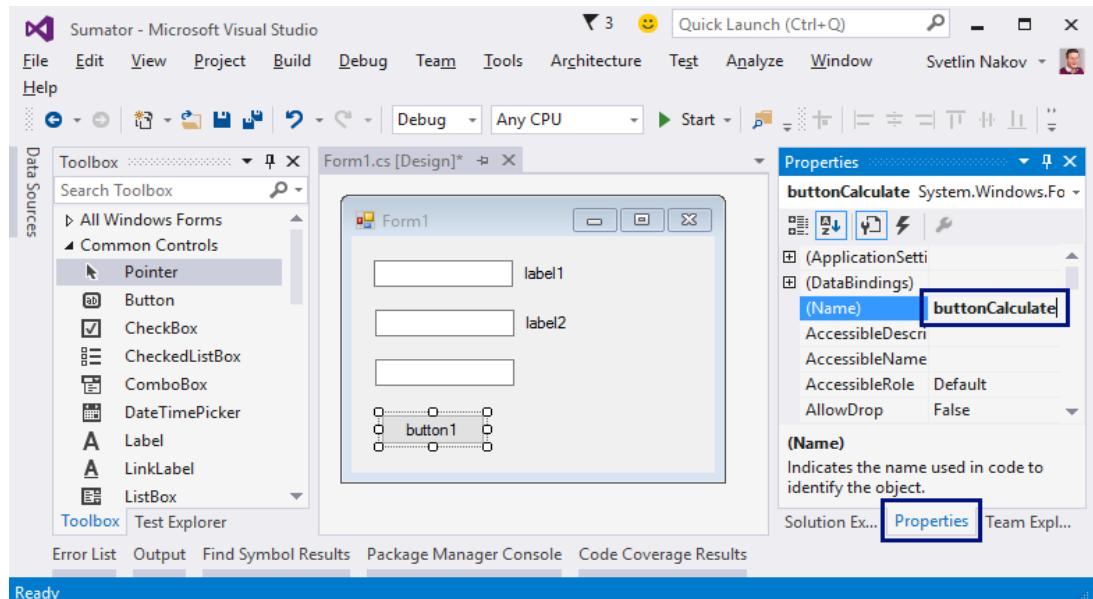
Ще създадем ново **Windows Forms приложение**, ще нарисуваме с редактора на Visual Studio **графичният интерфейс на приложението** (полетата, бутооните и останалите визуални елементи), ще дадем подходящи **имена за контролите**, след това ще **напишем кода**, който се изпълнява при натискане на бутона за сумиране на числата (ще използваме поредица от C# команди, каквито вече писахме в тази глава многократно), а накрая ще **прихванем възможните грешки**, за да покажем съобщение за грешка. Да започваме!

Във Visual Studio създаваме **нов C# проект от типа „Windows Forms Application“**.

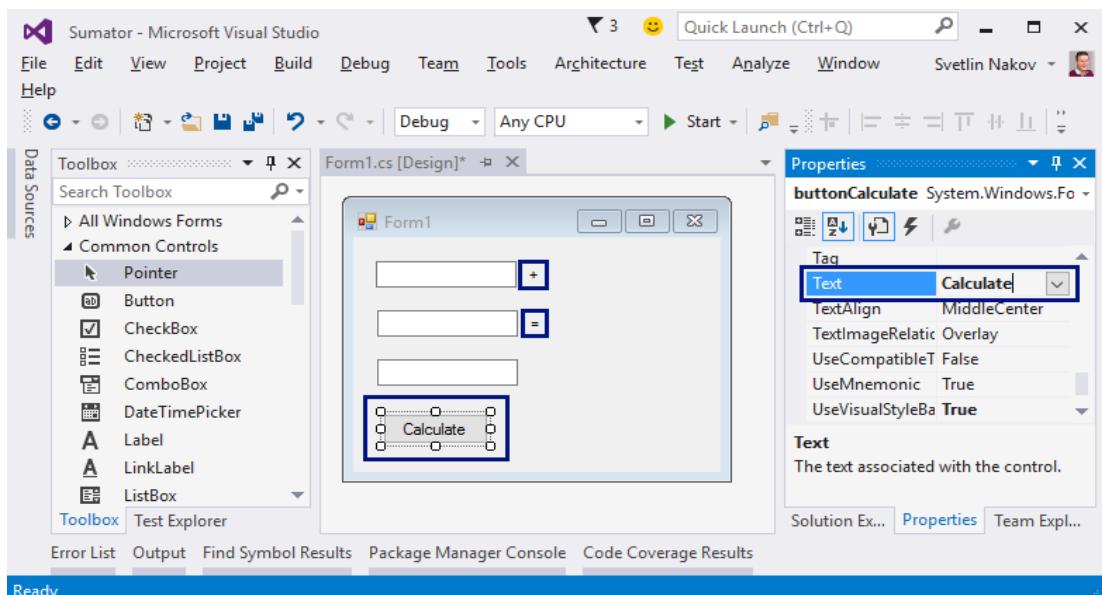
След създаването на Windows Forms приложение ще се появи **редактор за потребителски интерфейс**, в който могат да се слагат **различни визуални елементи**, например текст, кутийки с текст, бутони, картички, падащи списъци, менюта:



Изтегляме от лентата вляво (Toolbox) **три текстови полета (TextBox)**, **два надписа (Label)** и **един бутона (Button)**, след което ги подреждаме в прозореца на приложението. След това променяме имената на всяка от контролите. Това става от прозорчето “Properties” вдясно, чрез промяна на полето (**Name**):



Продължаваме да **настройваме контролите** във формата, използвайки Windows Forms редактора на Visual Studio:



Задаваме следните **имена на графичните елементи**:

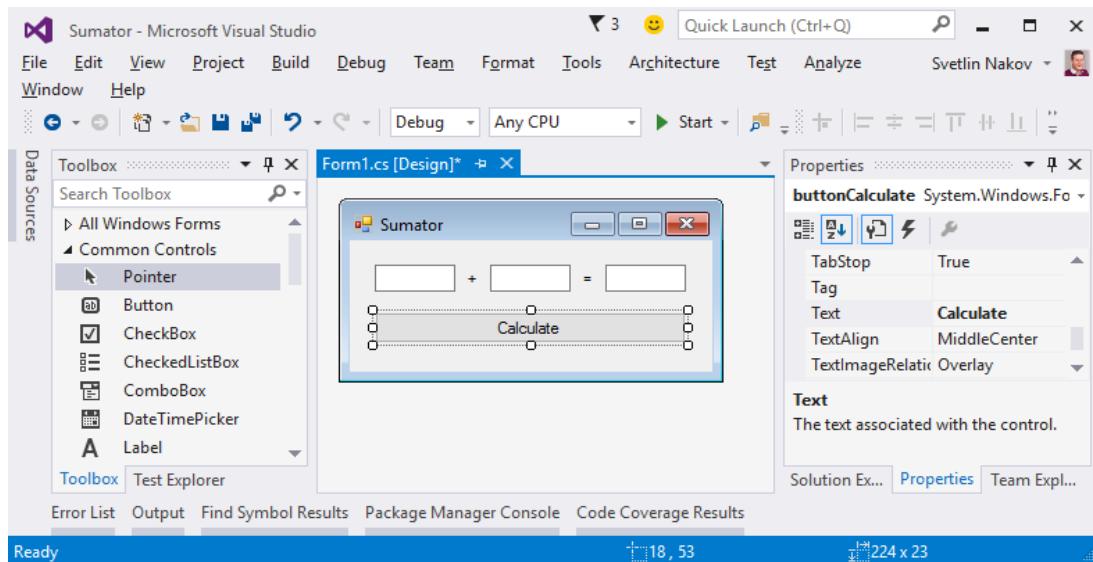
- Имена на текстовите полета: **textBox1**, **textBox2**, **textBoxSum**
- Име на бутона: **buttonCalculate**

- Име на формата: **FormCalculate**

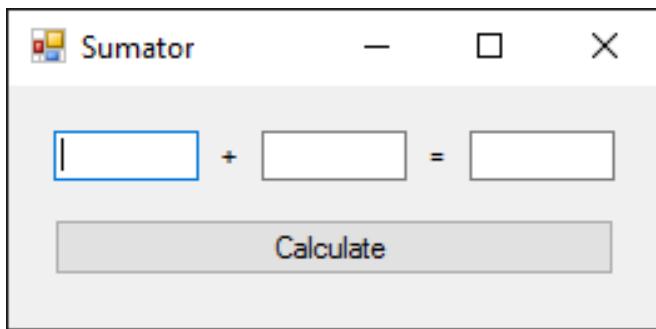
Променяме заглавията (**Text** свойството) на контролите:

- Form1** → “**Sumator**”; **buttonCalculate** → “**Calculate**”
- label1** → “**+**”; **label2** → “**=**”

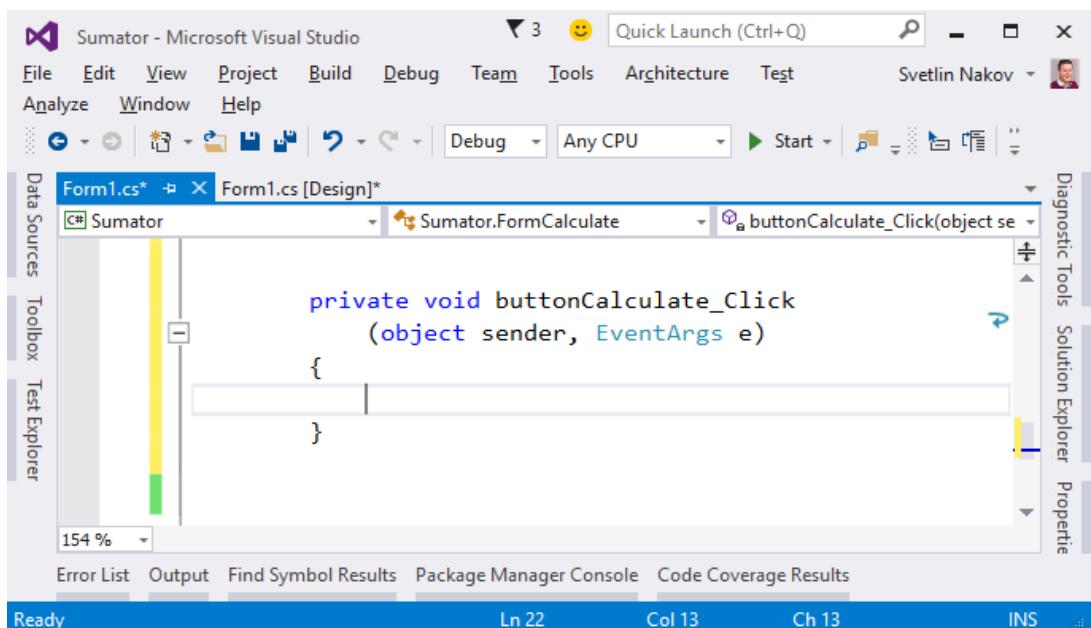
Преоразмеряваме и подреждаме контролите, за да изглеждат по-добре:



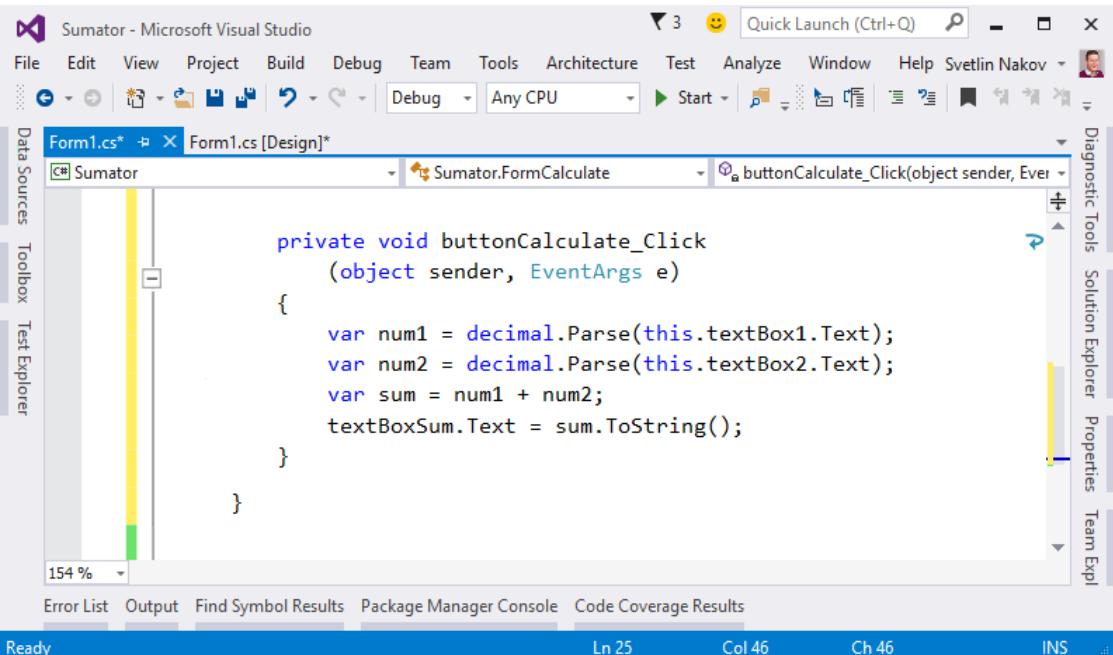
Опитваме да пуснем приложението с **[Ctrl+F5]**. То би трябвало да стартира, но да не функционира напълно, защото не сме написали какво се случва при натискане на бутона **[Calculate]**.



Сега е време да напишем кода, който **сумира числата** от първите две полета и показва **резултата** в третото поле. За целта кликваме **два пъти върху бутона [Calculate]**. Ще се появи място в C# кода, в което да напишем какво да се случва при натискане на бутона:

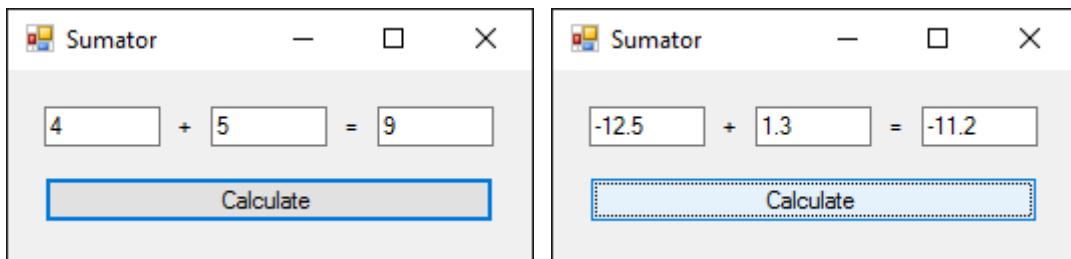


Написваме следния C# код между отварящата и затварящата скоба {}, точно където стои първоначално курсорът:

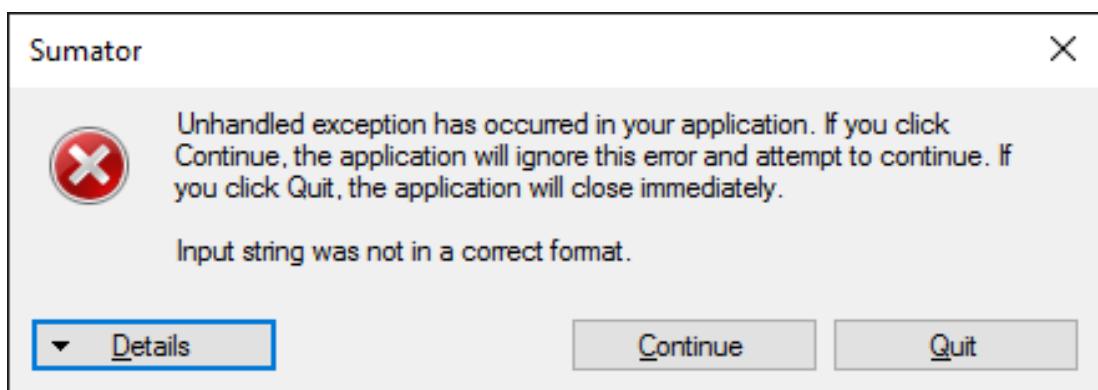
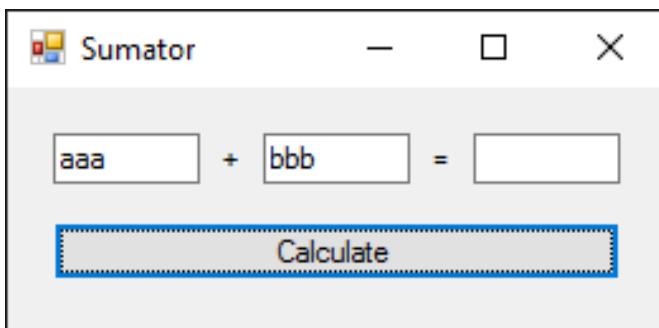


Този код взима първото число от полето **textBox1** и го запазва в променливата **num1**, запазва второто число от полето **textBox2** в променливата **num2**, след това сумира **num1** и **num2** в променливата **sum** и накрая извежда текстовата стойност на променливата **sum** в полето **textBoxSum**.

Стартираме отново програмата с [Ctrl+F5] и проверяваме дали работи коректно. Правим опит да сметнете $4 + 5$, а след това $-12.5 + 1.3$:



Пробваме и с **невалидни числа**, напр. "aaa" и "bbb". Изглежда има проблем:



Проблемът идва от прехвърлянето на текстово поле в число. Ако стойността в полето **не е число**, програмата дава грешка (exception). Можем да поправим кода, за да коригираме този проблем:

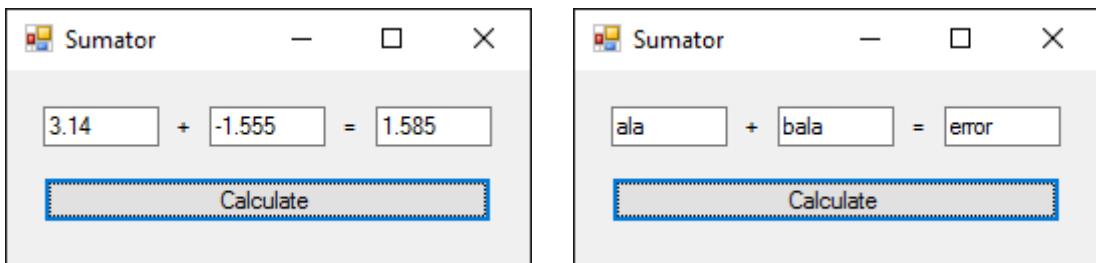
```
private void buttonCalculate_Click
    (object sender, EventArgs e)
{
    try
    {
        var num1 = decimal.Parse(this.textBox1.Text);
        var num2 = decimal.Parse(this.textBox2.Text);
        var sum = num1 + num2;
```

```

    textBoxSum.Text = sum.ToString();
}
catch (Exception)
{
    textBoxSum.Text = "error";
}
}

```

Горният код прихваща грешките при работа с числа (хваща изключенията) и в случай на грешка извежда стойност **error** в полето с резултата. Стартираме отново програмата с [Ctrl+F5] и я пробваме дали работи. Този път при грешно число резултатът е **error** и програмата не се чупи:



Сложно ли е? Нормално е да е сложно, разбира се. Тъкмо започваме да навлизаме в програмирането. Примерът по-горе изиска още много знания и умения, които ще развиваме в тази книга и даже и след нея. Просто си позволете да се позабавлявате с desktop програмирането.

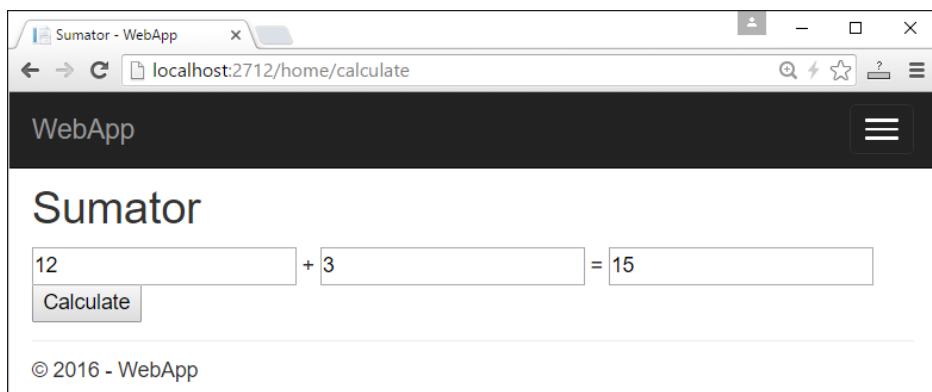
Ако не тръгва нещо, гледайте [видеото в началото на тази глава](#) или питайте във форума на СофтУни: <https://softuni.bg/forum>. Или продължете смело напред към следващия пример или към следващата глава от книгата. Ще дойде време и ще ви е лесно, но наистина трябва да вложите **усърдие и постоянство**. Програмирането се учи бавно и с много, много практика.

Уеб приложение: суматор за числа

Сега ще напишем нещо още по-сложно, но и по-интересно: уеб приложение, кое-то **изчислява сумата на две числа**.

По идея приложението трябва да се отваря в стандартен **уеб браузър** и при желание може да се качи (deploy) в Интернет. При **въвеждане на две числа** в първите две текстови полета и натискане на бутона **[Calculate]** се **изчислява тяхната сума** и резултатът се показва в третото текстово поле.

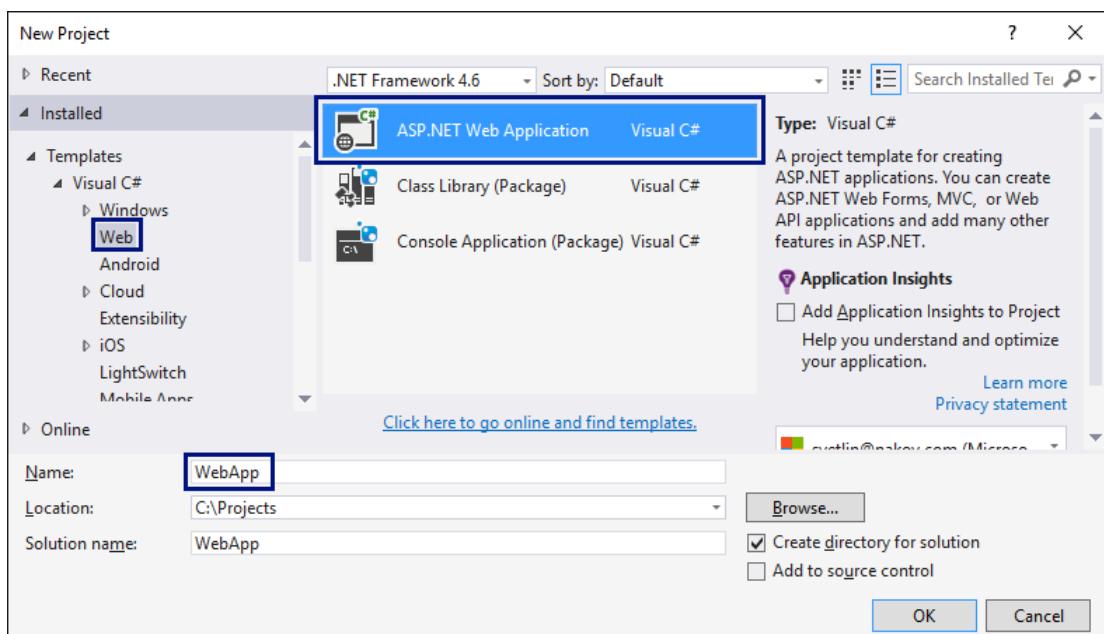
Обърнете внимание, че ще създадем **уеб-базирано приложение**. Това е приложение, което е достъпно през уеб браузър, точно както любимата ви уеб поща или новинарски сайт. Ето как може да изглежда уеб приложението в действие:



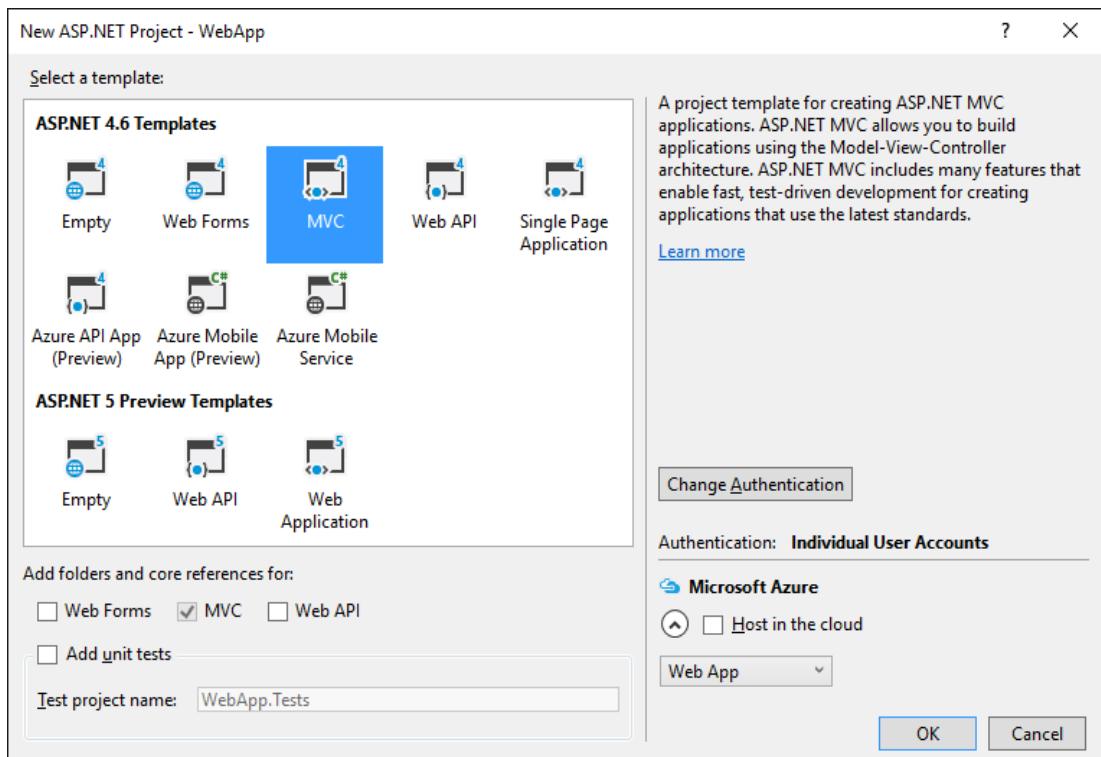
Уеб приложението ще има сървърна част (back-end), която е написана на езика C# с технологията ASP.NET MVC и клиентска част (front-end), която е написана на езика HTML (това е език за визуализация на информация в уеб браузър). Уеб приложението се очаква да изглежда приблизително като на картиинката по-горе.

За разлика от конзолните приложения, които четат и пишат данните си във вид на текст на конзолата, уеб приложениета имат **уеб базиран потребителски интерфейс**. Уеб приложениета се **зареждат от някакъв Интернет адрес** (URL) чрез стандартен уеб браузър. Потребителите пишат входните данни в страница, визуализирана от уеб браузъра, данните се обработват на уеб сървър и резултатите се показват отново в страницата в уеб браузъра. За нашето уеб приложение ще използваме **технологията ASP.NET MVC**, която позволява създаване на **уеб приложения с езика за програмиране C#** в средата за разработка **Visual Studio**.

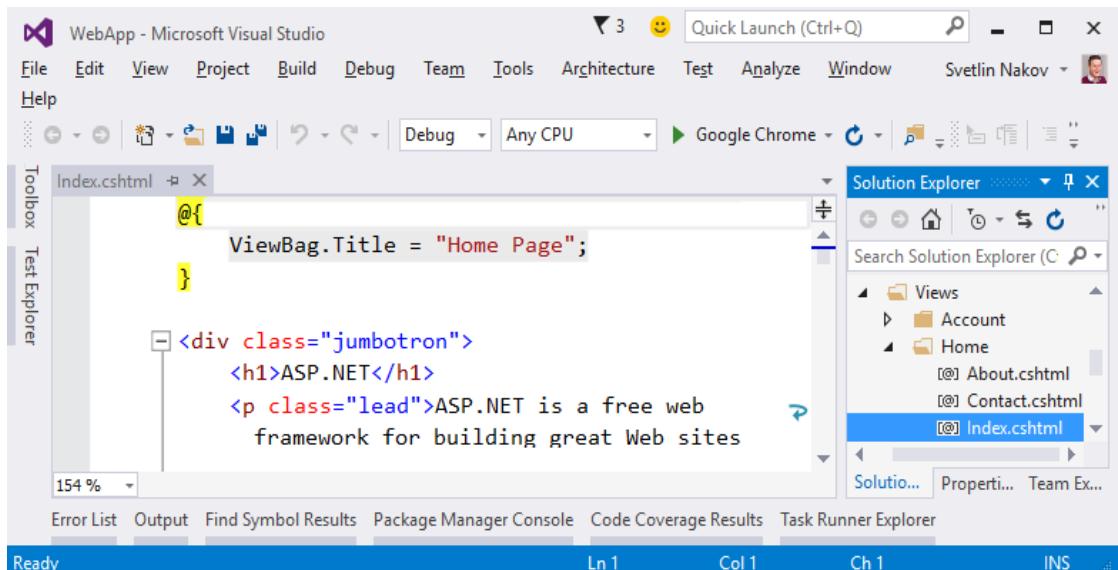
Във Visual Studio създаваме нов C# проект от тип „ASP.NET Web Application“ с име **WebApp**:



Избираме **ТИП** на приложението – “**MVC**”:



Намираме файла **Views\Home\Index.cshtml**. В него се намира **изгледът (view)** за **главната страница** на нашето уеб приложението:



Изтриваме стария код от файла **Index.cshtml** и пишем следния код:

```

@{
    ViewBag.Title = "Sumator";
}

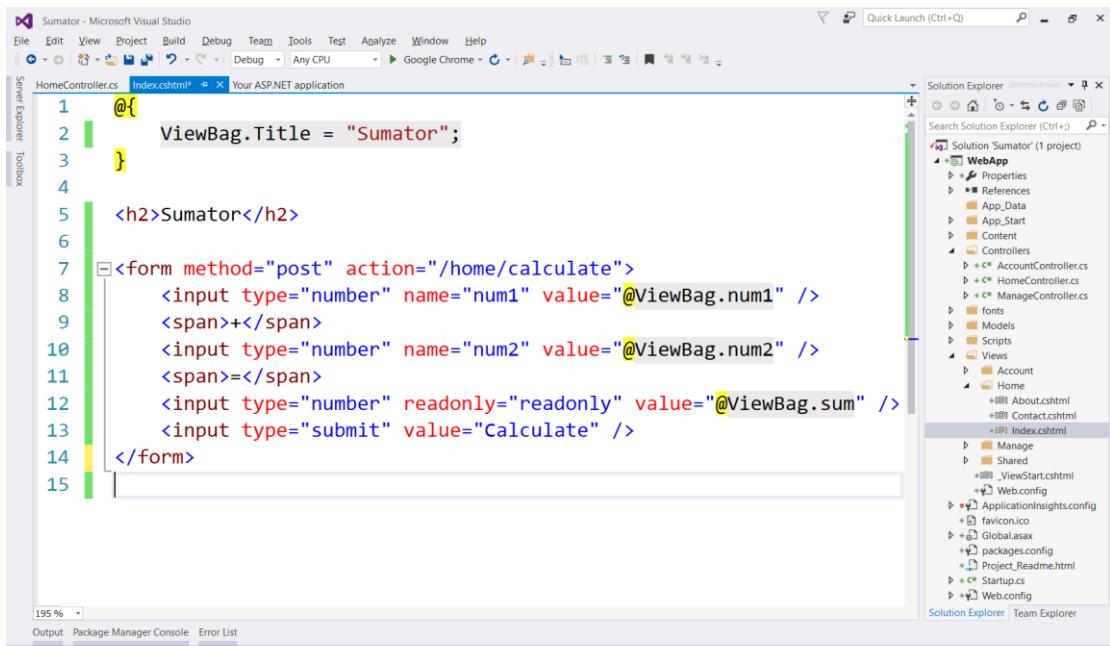
<h2>Sumator</h2>

<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>

```

Този код създава една уеб форма с три текстови полета и един бутона в нея. В полетата се зареждат стойности, които се изчисляват предварително в обекта **ViewBag**. Указано е, че при натискане на бутона [Calculate] ще се извика действието **/home/calculate** (действие **calculate** от **home** контролера).

Ето как трябва да изглежда файлът **Index.cshtml** след промяната:

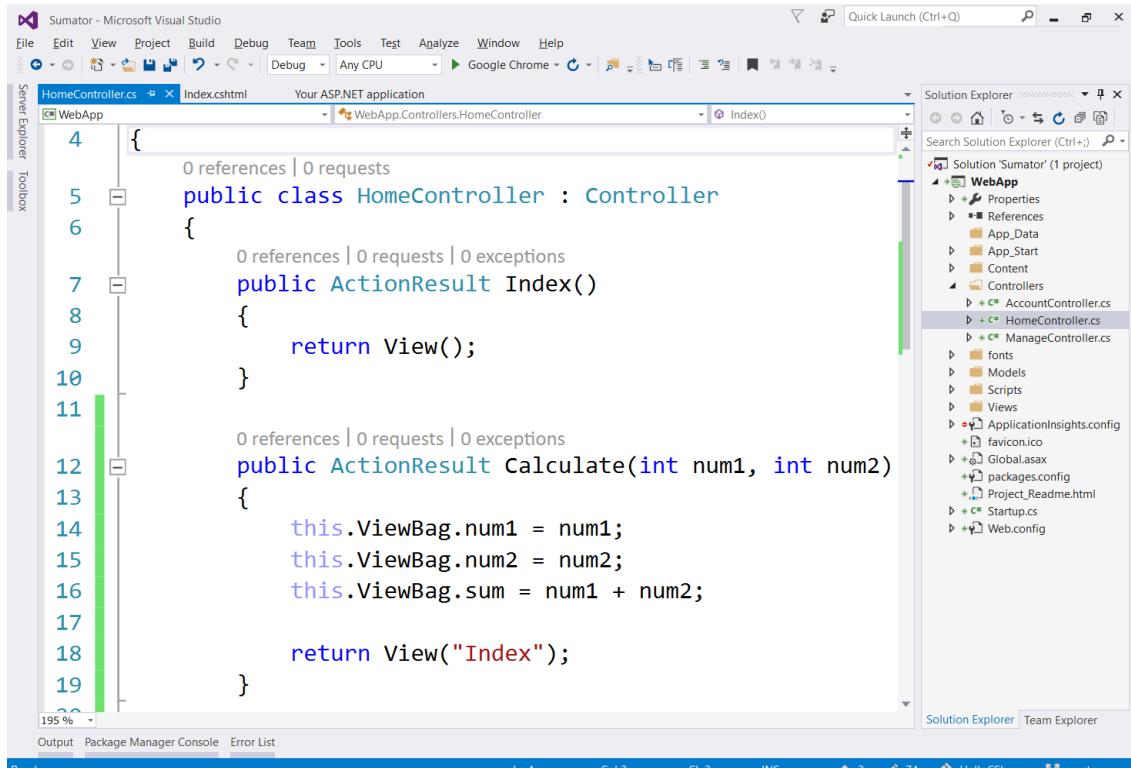


Остава да се напише **действието** (**action**), което сумира числата при натискане на бутона [Calculate]. Отваряме файла **Controllers\HomeController.cs** и добавяваме следния код в тялото на **HomeController** класа:

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

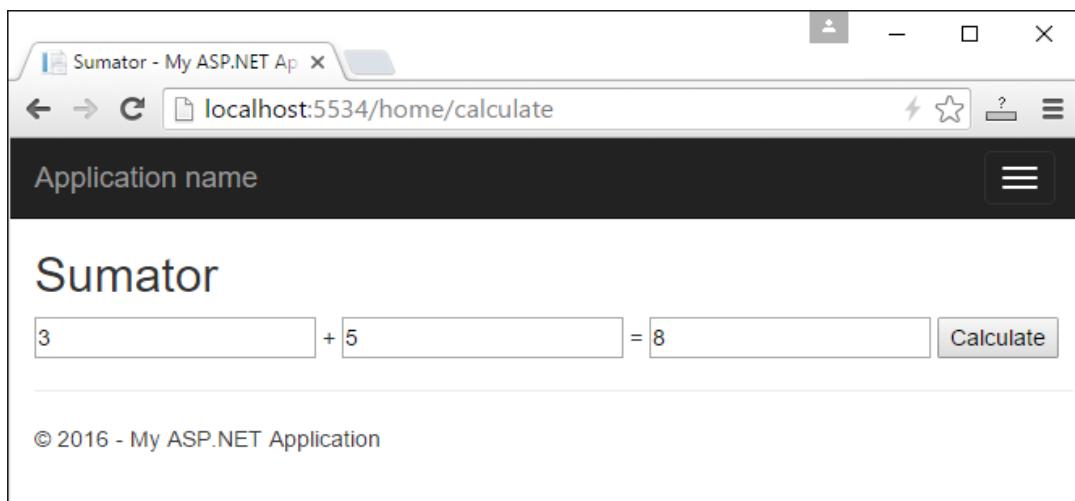
Този код осъществява действието "calculate". То приема два параметъра **num1** и **num2** и ги записва в обекта **ViewBag**, след което изчислява и записва тяхната сума. Записаните във **ViewBag** стойности след това се използват от изгледа, за да се покажат в **тритекстови полета** във формата за сумиране на числа в уеб страницата от приложението.

Ето как трябва да изглежда файлът **HomeController.cs** след промяната:



Уеб приложението за сумиране на числа е вече **готово**. Написахме **изгледът** (view), който рисува формата с данните в нея. Написахме и **контролера** (controller), който изпълнява действията извикани от формата.

Можем да го стартираме уеб приложението с **[Ctrl+F5]** и да тестваме дали работи както очакваме. След кратко забавяне за компилация и старт ще се отвори уеб браузърт по подразбиране и ще трябва да се визуализира следната страница:



Страшно ли изглежда? **Не се плащете!** Имаме да учим още много, за да достигнем ниво на знания и умения, за да пишем свободно уеб-базирани приложения, като в примера по-горе и много по-големи и по-сложни. Ако не успеете да се справите, **няма страшно**, продължете спокойно напред. След време ще си спомняте с усмивка колко непонятен и вълнуващ е бил първият ви сблъсък с уеб програмирането.

Ако имате проблеми с примера по-горе, **гледайте видеото** в началото на тази глава. Там приложението е направено на живо стъпка по стъпка с много обяснения. Или питайте във **форума на СофтУни**: <https://softuni.bg/forum>.

Целта на горните два примера (графично desktop приложение и уеб приложение) не е да се научите да правите такива приложения, а да се докоснете по-надълбоко до програмирането, **да разпалите интереса си** към разработката на софтуер и да се вдъхновите да учите по-сериозно. С времето ще се научите не само да програмирате прости конзолни програмки, но и да създавате по-големи настолни приложения, уеб сайтове и приложения, мобилни апликации, приложения за домашните ви уреди и много други.

Имате да учите още много, но пък е интересно, нали? Отдъхнете малко и продължаваме със следващата тема от основите на програмирането.

Глава 2.1. Прости пресмятания с числа

В настоящата глава ще се запознаем със следните концепции и програмни техники за писане на C# програми:

- Какво представлява **системната конзола**?
- Как да **прочитаме числа** от системната конзола?
- Как да работим с **типове данни и променливи**, които са ни необходими при обработка на числа и операциите между тях?
- Как да **изведем** резултат (число) на системната конзола?
- Как да извършваме прости **аритметични операции**: събиране, изваждане, умножение, деление, съединяване на низ?

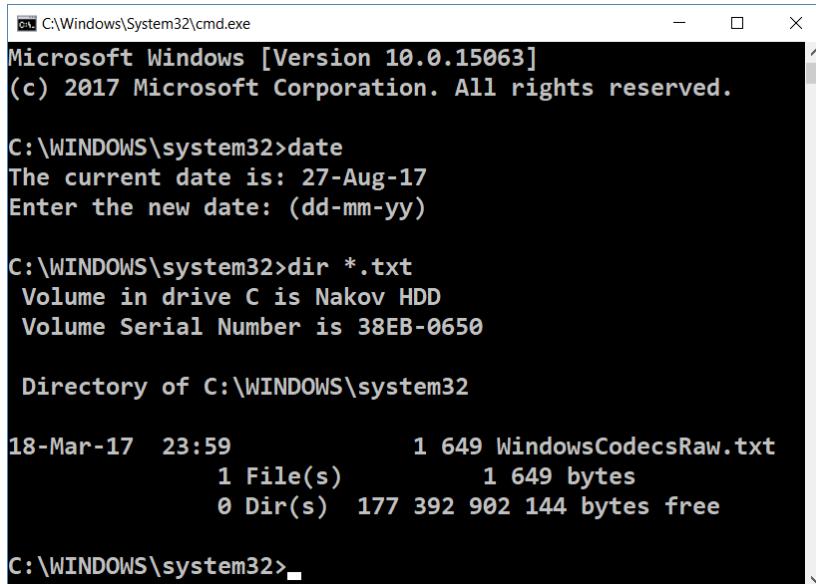
Видео

Гледайте видео-урок по тази глава: <https://youtube.com/watch?v=Of7c9RIZGaE>.

Системна конзола

Обикновено наричана само "**конзола**", системната, или още компютърната конзола, представлява устройството, чрез което подаваме **команди** на компютъра в текстов вид и получаваме резултатите от тяхното изпълнение отново като текст.

В повечето случаи системната конзола представлява текстов терминал, т.е. приема и визуализира само **текст**, без графични елементи като например бутони, менюта и т.н. Обикновено изглежда като прозорец с черен цвят като този:



```
C:\Windows\System32\cmd.exe
Microsoft Windows [Version 10.0.15063]
(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)  177 392 902 144 bytes free

C:\WINDOWS\system32>
```

В повечето операционни системи **конзолата** е достъпна като самостоятелно приложение на което пишем конзолни команди. В Windows се нарича **Command Prompt**, а в Linux и Mac се нарича **Terminal**. В конзолата се изпълняват конзолни

приложения. Те четат текстов вход от командния ред и печатат изхода си като текстов изход на конзолата. В настоящата книга ще се учит на програмиране като създаваме предимно **конзолни приложения**.

Четене на числа от конзолата

За да прочетем **цяло** (не дробно) **число** от конзолата е необходимо да **декларираме променлива**, да посочим **типа на числото**, както и да използваме стандартната команда за четене на информация от системната конзола:

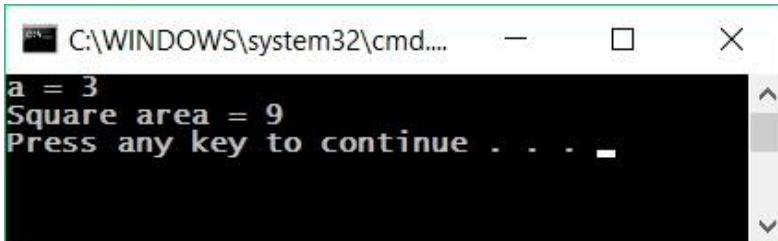
```
var num = int.Parse(Console.ReadLine());
```

Пример: пресмятане на лице на квадрат със страна a

За пример да вземем следната програма, която прочита цяло число от конзолата, умножава го по него самото (вдига го на квадрат) и отпечатва резултата от умножението. Така можем да пресметнем лицето на квадрат по дадена дължина на страната:

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

Ето как би работила програмата при квадрат с размер на страната 3:



Опитайте да въведете грешно число, например "hello". Ще получите съобщение за грешка по време на изпълнение (exception). Това е нормално. По-късно ще разберем как можем да прихващаме такива грешки и да караме потребителят да въвежда число наново.

Как работи примерът?

Първият ред `Console.WriteLine("a = ");` печата информативно съобщение, което подканва потребителя да въведе страната на квадрата **a**. След отпечатването курсорът остава на същия ред. Оставането на същия ред е по-удобно за потребителя, чисто визуално. Използва се `Console.WriteLine(...)`, а не `Console.WriteLine(...)` и така курсорът остава на същия ред.

Следващият ред `var a = int.Parse(Console.ReadLine());` прочита цяло число от конзолата. Възможно първо се прочита текст (стринг) чрез `Console.ReadLine()` и след това се преобразува до цяло число чрез `int.Parse(...)`. Резултатът се записва в променлива с име `a`.

Следващата команда `var area = a * a;` записва в нова променлива `area` резултата от умножението на `a` по `a`.

Следващата команда `Console.WriteLine("Square area = ");` отпечатва посочения текст, без да преминава на нов ред. Отново се използва `Console.WriteLine(...)`, а не `Console.WriteLine()` и така курсорът остава на същия ред, за да може след това да се отпечата и изчисленото лице на квадрата.

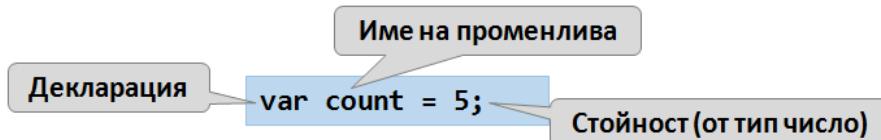
Последната команда `Console.WriteLine(area);` отпечатва изчислената стойност от променливата `area`.

Тестване в Judge системата

Тествайте решението си: <https://judge.softuni.bg/Contests/Practice/Index/504#0>.

Пресмятания в програмирането

За компютрите знаем, че са машини, които обработват данни. Всички **данни** се записват в компютърната памет (RAM памет) в **променливи**. Променливите са именувани области от паметта, които пазят данни от определен тип, например число или текст. Всяка една **променлива** в C# има **име**, **тип** и **стойност**. Ето как бихме дефинирали една променлива, като едновременно с декларацията ѝ, ѝ присвояваме и стойност:



След тяхната обработка, данните се записват отново в променливи (т.е. някъде в паметта, заделена от нашата програма).

Типове данни и променливи

В програмирането всяка една променлива съхранява определена **стойност** от даден **тип**. Типовете данни могат да бъдат например: **число**, **буква**, **текст** (стринг), **дата**, **цвят**, **картичка**, **списък** и др. Ето няколко примера за типове данни:

- тип **цяло число**: 1, 2, 3, 4, 5, ...
- тип **дробно число**: 0.5, 3.14, -1.5, ...
- тип **буква от азбуката** (символ): 'a', 'b', 'c', ...
- тип **текст** (стринг): "Здрави", "Hi", "Beer", ...
- тип **ден от седмицата**: Monday, Tuesday, ...

Четене на дробно число от конзолата

За да прочетем **дробно число** от конзолата е необходимо отново да **декларираме променлива**, да посочим **типа на числото**, както и да използваме стандартната команда за четене на информация от системната конзола:

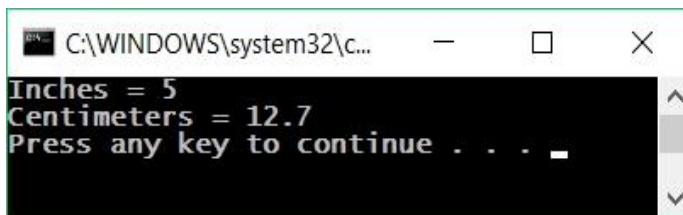
```
var num = double.Parse(Console.ReadLine());
```

Пример: прехвърляне от инчове в сантиметри

Да напишем програма, която чете дробно число в инчове и го обръща в сантиметри:

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

Да стартираме програмата и да се уверим, че при подаване на стойност в инчове, получаваме коректен резултат в сантиметри:



Тестване в Judge системата

Тествайте решението си: <https://judge.softuni.bg/Contests/Practice/Index/504#1>

Четене и печтане на текст

За да прочетем текст (стринг) от конзолата, отново **декларираме нова променлива** и използваме стандартната **команда за четене на информация от системната конзола**:

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

Нека обърнем внимание на факта, че при **четене на текст не се декларира** по никакъв начин тип **"string"** (текст). Това е така, защото по подразбиране методът **Console.ReadLine(...)** връща като **результат текст**. Допълнително, вие можете да зададете текста да бъде прехвърлен в цяло число чрез **int.Parse(...)** или дробно число чрез **double.Parse(...)**. Ако това не се направи, за програмата **всяко едно**

число ще бъде просто **текст**, с който не бихме могли да извършваме аритметични операции.

Пример: поздрав по име

Да напишем програма, която въвежда името на потребителя и го поздравява с текста "Hello, {име}".

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

В този случай, изразът **{0}** е заместен от **първия** подаден аргумент, който в примера е променливата **name**:

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

Тестване в Judge системата

Тествайте решението си: <https://judge.softuni.bg/Contests/Practice/Index/504#2>

Съединяване на текст и числа

При печат в конзолата на текст, числа и други данни, **можем да ги съединим**, като използваме шаблони **{0}**, **{1}**, **{2}** и т.н. В програмирането тези шаблони се наричат **placeholders**.

```
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);
```

Ето резултатът, който ще получим, след изпълнение на този пример:

```
C:\WINDOWS\system32\cmd.exe
Ivan
Ivanov
30
Plovdiv
You are Ivan Ivanov, a 30-years old person from Plovdiv.
Press any key to continue . . .
```

Обърнете внимание как всяка една променлива трябва да бъде подадена в реда, в който искаме да се печата. По същество, шаблонът (placeholder) приема променливи от всякакъв вид.

Възможно е един и същ номер на шаблон да се използва по няколко пъти и не е задължително шаблоните да са номерирани поредно. Ето пример:

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

Резултатът е:

```
1 + 1 = 2
```

Тестване в Judge системата

Тествайте решението си: <https://judge.softuni.bg/Contests/Practice/Index/504#3>.

Аритметични операции

Да разгледаме базовите аритметични операции в програмирането.

Събиране на числа (оператор +)

Можем да събираме числа с оператора `+`:

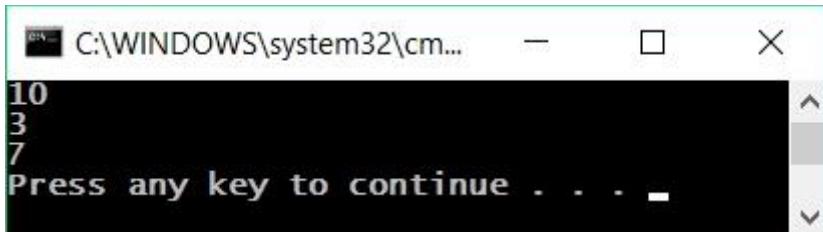
```
var a = 5;
var b = 7;
var sum = a + b; // резултатът е 12
```

Изваждане на числа (оператор -)

Изваждането на числа се извършва с оператора `-`:

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

Ето резултатът от изпълнението на програмата (при числа 10 и 3):



Умножение на числа (оператор *)

За умножение на числа използваме оператора *****:

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

Деление на числа (оператор /)

Делението на числа се извършва с оператора **/**. Той работи различно при цели и при дробни числа.

- Когато делим две цели числа, се извършва **целочислено деление** и полученият резултат е цяло число с отрязана дробна част. Например $11 / 3 = 3$.
- Когато делим две числа, от които поне едното е дробно, се извършва **дробно деление** и полученият резултат е дробно число, както в математиката. Например $11 / 4.0 = 2.75$. При невъзможност за точно разделяне, резултатът се закръгля, например $11.0 / 3 = 3.66666666666667$.
- Целочисленото **деление на 0** предизвиква **грешка** по време на изпълнение (runtime exception).
- Дробното **деление на 0** не предизвиква грешка, а резултатът е **+/- безкрайност** или специалната стойност **NaN**. Например $5 / 0.0 = \infty$.

Ето няколко примера за използване на оператора за делене:

```
var a = 25;
var i = a / 4; // извършваме т.нр. целочислено деление:
результатът от тази операция ще бъде 6, защото дробната част се
отрязва, тъй като извършваме деление с цели числа

var f = a / 4.0; // 6.25 - дробно деление. Изрично сме
указали числото 4 да се интерпретира като дробно, като сме
добавили десетичната точка, следвана от нула

var error = a / 0; // Грешка: целочислено деление на 0
```

Да разгледаме и няколко примера за **целочислено деление** (запомнете, че при **деление на цели числа** в езика C# резултатът е **цяло число**):

```
var a = 25;
Console.WriteLine(a / 4); // Целочислен резултат: 6
Console.WriteLine(a / 0); // Грешка: деление на 0
```

Да разгледаме няколко примера за **деление на дробни числа**. При дробно делене резултатът винаги е **дробно число** и деленето никога не дава грешка и работи коректно със специалните стойности **$+\infty$** и **$-\infty$** :

```
var a = 15;
Console.WriteLine(a / 2.0);    // Дробен резултат: 7.5
Console.WriteLine(a / 0.0);    // Резултат: Infinity
Console.WriteLine(-a / 0.0);   // Резултат: -Infinity
Console.WriteLine(0.0 / 0.0);   // Резултат: NaN (Not a Number),
т.е. резултатът от операцията не е валидна числена стойност
```

При отпечатването на стойностите **∞** и **$-\infty$** на конзолата може да излязат **?**, защото конзолата в Windows не поддържа коректно Unicode и поврежда повечето нестандартни символи, букви и специални знаци. Горният пример най-вероятно ще изведе следния **резултат**:

```
7.5
?
-
NaN
```

Съединяване на текст и число

Операторът **+** освен за събиране на числа служи и за съединяване на текст (долепяне на два символни низа един след друг). В програмирането съединяване на текст с текст или с число наричаме "**конкатенация**". Ето как можем да съединяваме текст и число с оператора **+**:

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

Ето още един пример:

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

Забелязвате ли нещо странно? Може би очаквахте числата **a** и **b** да се сумират? Въщност конкатенацията работи отляво надясно и горният резултат е абсолютно коректен. Ако искаме да сумираме числата, ще трябва да ползваме **скоби**, за да променим реда на изпълнение на операциите:

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

Числени изрази

В програмирането можем да пресмятаме и **числови изрази**, например:

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

В сила е стандартното правило за приоритетите на аритметичните операции: **умножение и деление се извършват винаги преди събиране и изваждане**. При наличие на израз в скоби, той се изчислява пръв, но ние знаем всичко това от училищната математика.

Пример: изчисляване на лице на трапец

Да напишем програма, която въвежда дължините на двете основи на трапец и неговата височина (по едно дробно число на ред) и пресмята **лицето на трапеца** по стандартната математическа формула:

```
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);
```

Ако стартираме програмата и въведем за страните съответно 3, 4 и 5, ще получим следния резултат:

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

Тестване в Judge системата

Тествайте решението си: <https://judge.softuni.bg/Contests/Practice/Index/504#4>.

Пример: периметър и лице на кръг

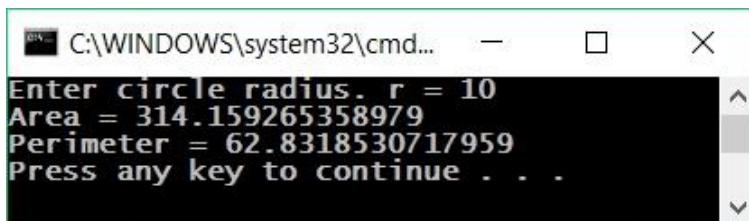
Нека напишем програма, която при въвеждане **радиуса r** на кръг **изчислява лицето и периметъра** на кръга / окръжността. Можем да използваме следните формули:

- Лице = $\pi * r * r$

- Периметър = $2 * \pi * r$
- $\pi \approx 3.14159265358979323846\ldots$

```
Console.WriteLine("Enter circle radius. r = ");
var r = double.Parse(Console.ReadLine());
Console.WriteLine("Area = " + Math.PI * r * r);
// Math.PI – вградена в C# константа за π
Console.WriteLine("Perimeter = " + 2 * Math.PI * r);
```

Да изprobваме програмата с радиус **r = 10**:

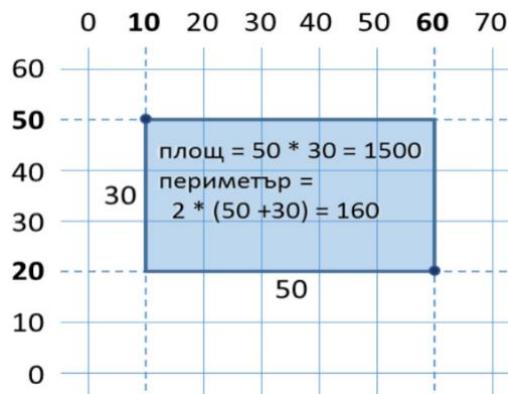


Тестване в Judge системата

Тествайте решението: <https://judge.softuni.bg/Contests/Practice/Index/504#5>.

Пример: лице на правоъгълник в равнината

Правоъгълник е зададен с координатите на два от своите два срещуположни ъгъла. Да се пресметнат площта и периметъра му:



В тази задача трябва да съобразим, че ако от по-големия **x** извадим по-малкия **x**, ще получим дължината на правоъгълника. Аналогично, ако от по-големия **y** извадим по-малкия **y**, ще получим височината на правоъгълника. Остава да умножим двете страни. Ето примерна имплементация на описаната логика:

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

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

```

var y2 = double.Parse(Console.ReadLine());
// Изчисляване страните на правоъгълника:
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));

```

Използваме **Math.Max(a, b)**, за да намерим по-голямата измежду стойностите **a** и **b** и аналогично **Math.Min(a, b)** за по-малката от двете стойности.

При стартиране на програмата със стойностите от координатната система в условието, получаваме следния резултат:

```

60
20
10
50
Area = 1500
Perimeter = 160
Press any key to continue . . .

```

Тестване в Judge системата

Тествайте решението си: <https://judge.softuni.bg/Contests/Practice/Index/504#6>

Какво научихме от тази глава?

Да резюмираме какво научихме от тази глава на книгата:

- Въвеждане на текст: `var str = Console.ReadLine();`.
- Въвеждане на цяло число: `var num = int.Parse(Console.ReadLine());`.
- Въвеждане на дробно число:
`var num = double.Parse(Console.ReadLine());`.
- Извършване на пресмятания с числа и използване на съответните аритметични оператори [+,-,*,/,:]: `var sum = 5 + 3;`.
- Извеждане на текст по шаблон на конзолата: `Console.WriteLine("{0} + {1} = {2}", 3, 5, 3 + 5);`.

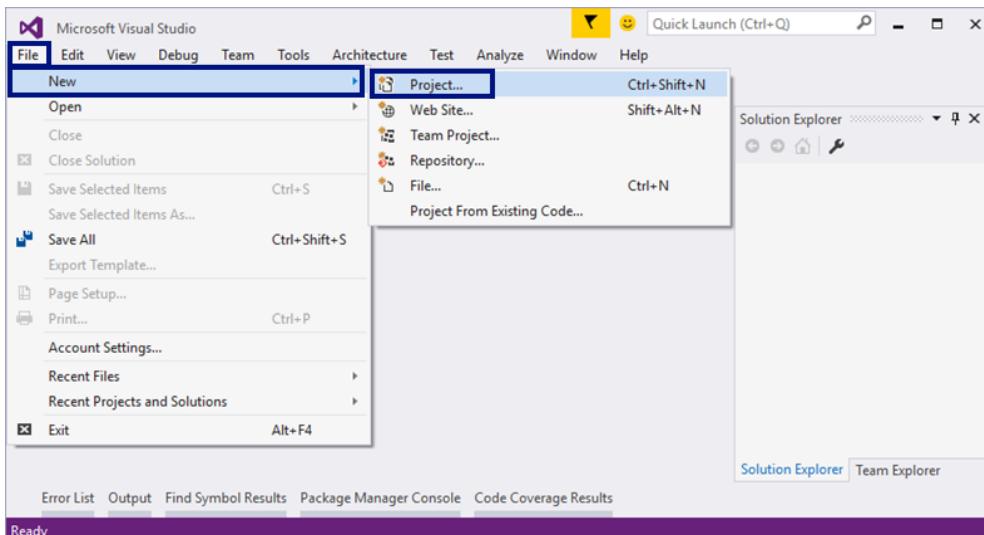
Упражнения: прости пресмятания

Нека затвърдим наученото в тази глава с няколко задачи.

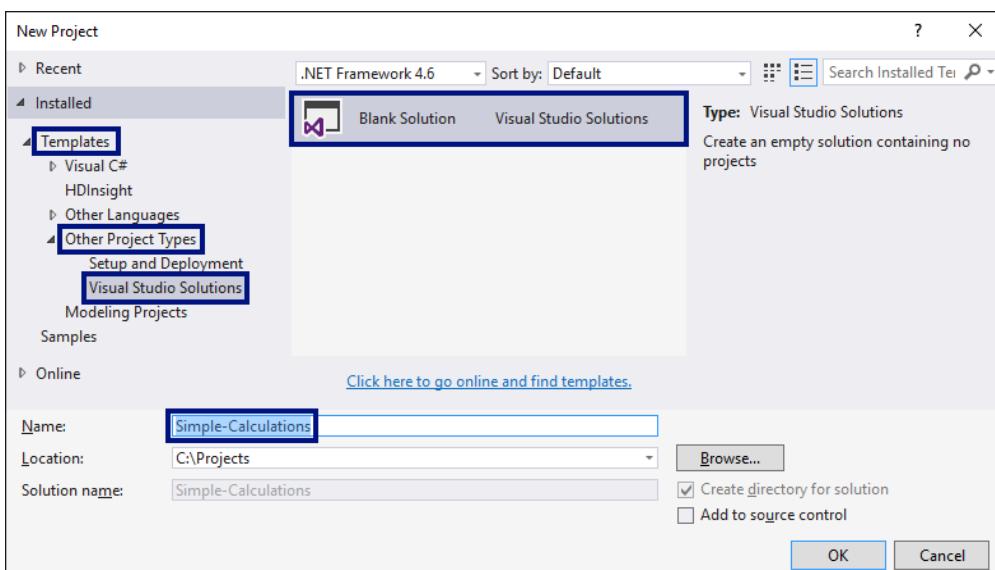
Празно Visual Studio решение (Blank Solution)

Започваме, като създадем празно решение (Blank Solution) във Visual Studio. Решенията (solutions) във Visual Studio обединяват **група проекти**. Тази възможност е **изключително удобна**, когато искаме да работим по **няколко проекта** и бързо да превключваме между тях или искаме да **обединим логически няколко взаимосвързани проекта**. В настоящото практическо занимание ще използваме **Blank Solution с няколко проекта**, за да организираме решенията на задачите от упражненията – всяка задача в отделен проект и всички проекти в общ solution.

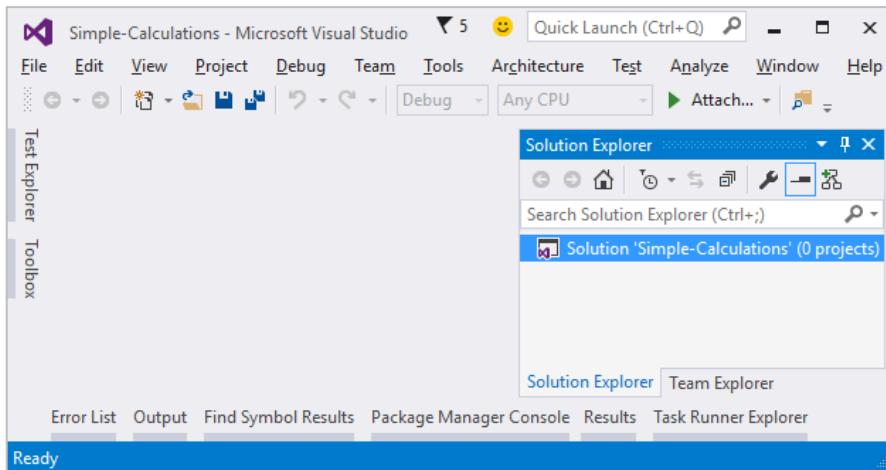
- Стартураме Visual Studio.
- Създаваме нов Blank Solution: [File] -> [New] -> [Project].



Избираме от диалоговия прозорец [Templates] -> [Other Project Types] -> [Visual Studio Solutions] -> [Blank Solution] и даваме подходящо име на проекта, например "Simple-Calculations":



Сега имаме създаден празен Visual Studio Solution (с 0 проекта в него):



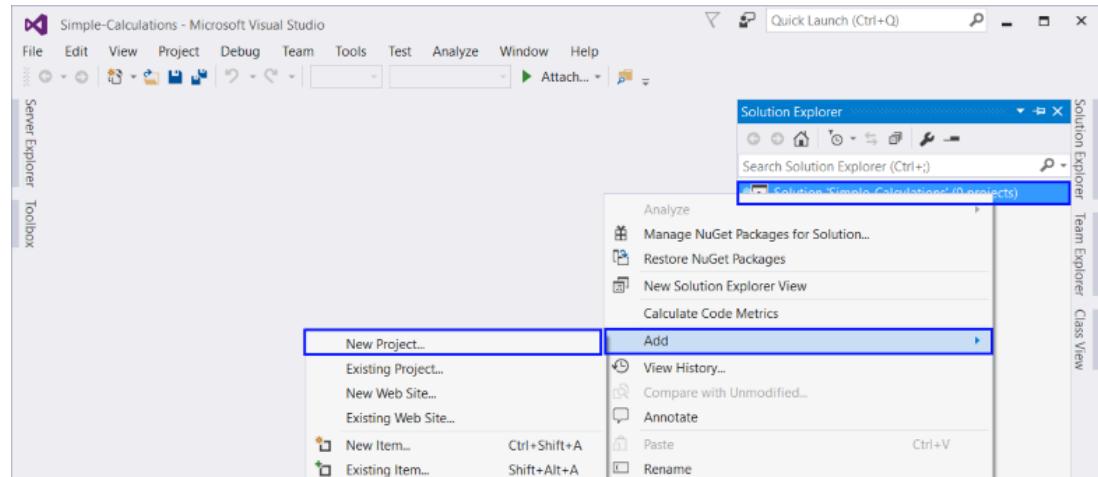
Целта на този **blank solution** е да добавяме в него по един проект за всяка задача от упражненията.

Задача: пресмятане на лице на квадрат

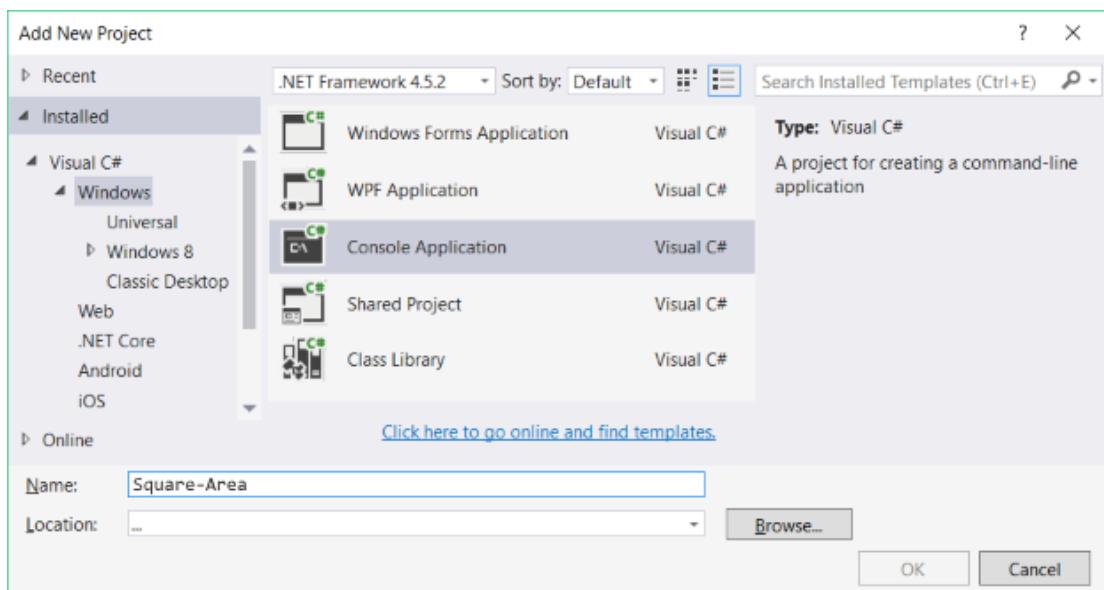
Първата задача от тази тема е следната: да се напише конзолна програма, която въвежда цяло число **a** и пресмята лицето на квадрат със страна **a**. Задачата е три-виално лесна: въвеждате число от конзолата, умножавате го само по себе си и печатате получения резултат на конзолата.

Насоки и подсказки

Създаваме **нов проект** в съществуващото Visual Studio решение. В Solution Explorer кликнете с десен бутон на мишката върху **Solution 'Simple-Calculations'**. Изберете [Add] -> [New Project...]:



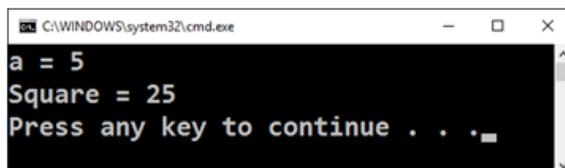
Ще се отвори **диалогов прозорец** за избор на **ТИП проект** за създаване. Избираме **C# конзолно приложение** с име "Square-Area":



Вече имаме solution с едно конзолно приложение в него. Остава да напишем кода за решаване на задачата. За целта отиваме в тялото на метода **Main(string[] args)** и пишем следния код:

```
namespace Square_Area
{
    0 references
    class Program
    {
        0 references
        static void Main(string[] args)
        {
            Console.Write("a = ");
            var a = int.Parse(Console.ReadLine());
            var area = a * a;
            Console.Write("Square = ");
            Console.WriteLine(area);
        }
    }
}
```

Кодът въвежда цяло число чрез **a = int.Parse(Console.ReadLine())**, след това изчислява **area = a * a** и накрая печата стойността на променливата **area**. Стартираме програмата с [Ctrl+F5] и я тестваме с различни входни стойности.



Тестване в Judge системата

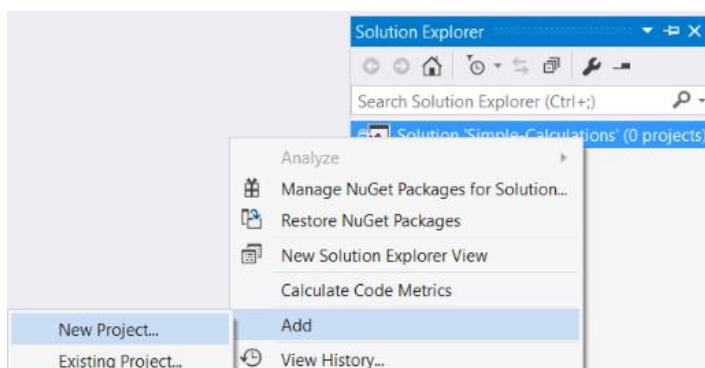
Тествайте решението си: <https://judge.softuni.bg/Contests/Practice/Index/504#0>. Трябва да получите 100 точки (напълно коректно решение):

| Submissions | | |
|-------------|----------------------------------|---------------------|
| Points | Time and memory used | Submission date |
| 100 / 100 | Memory: 7.77 MB Time: 0.013 s | 20:06:36 21.01.2016 |

Задача: от инчове към сантиметри

Да се напише програма, която чете от конзолата число (не непременно цяло) и преобразува числото от инчове в сантиметри. За целта умножава инчовете по 2.54 (зашщото 1 инч = 2.54 сантиметра).

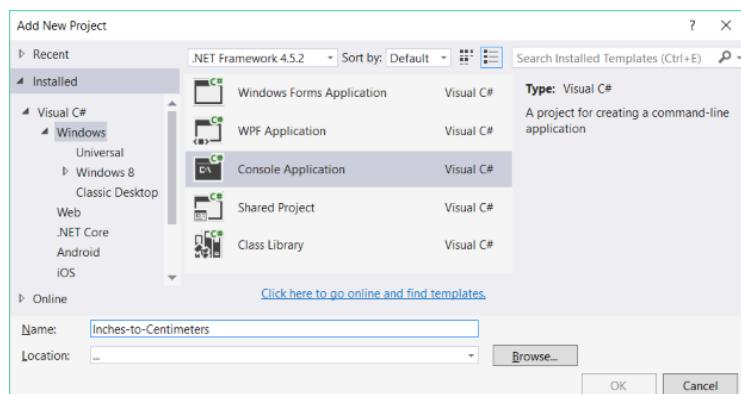
Насоки и подсказки



Първо създаваме нов C# конзолен проект в решението "Simple-Calculations". Кликаме с мишката върху решението в Solution Explorer и избираме [Add] -> [New Project...].

Избираме [Visual C#] -> [Windows] -> [Console Application] и задаваме

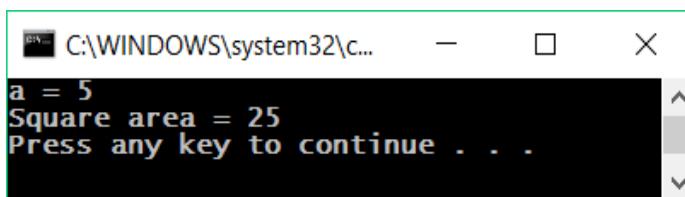
подходящо име, например "Inches-to-Centimeters":



Следва да напишем кода на програмата:

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

Стартираме програмата с [Ctrl+F5]:

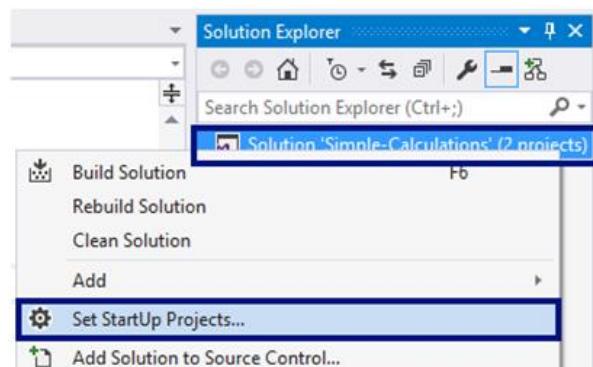


Изненада! Какво става? Програмата не работи правилно... Как така?

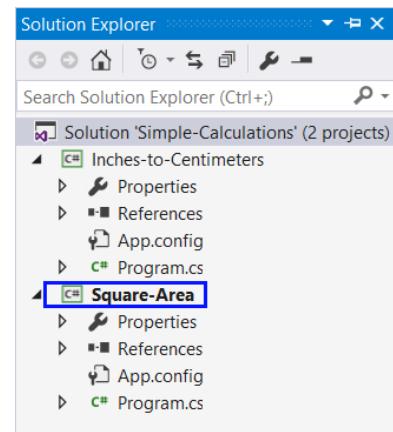
Възможност това не е ли предходната програма?

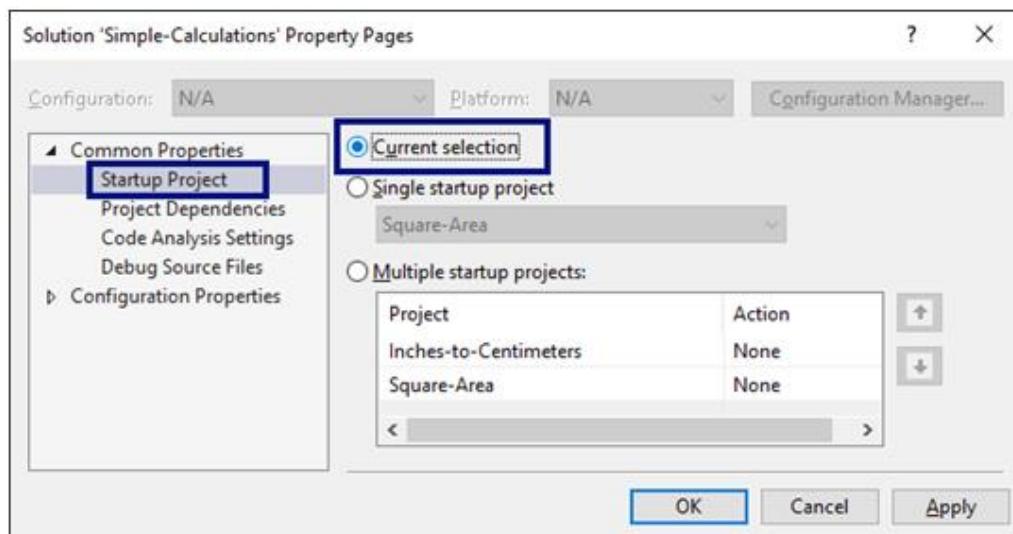
Знаехте ли, че във Visual Studio **текущият активен проект** в един solution е маркиран в получерно и може да се сменя (вж. картинката вдясно)?

За да включим режим на **автоматично преминаване към текущия проект**, кликаме върху главния solution (най-горният ред в Solution Explorer) с десен бутон на мишката и избираме [Set StartUp Projects...]:

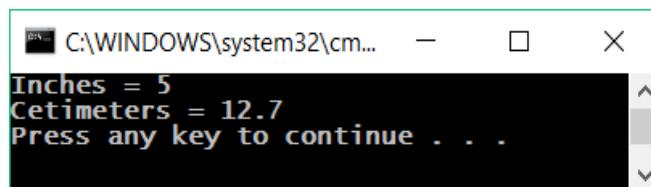


Ще се появи диалогов прозорец, от който трябва да се избере [Startup Project] -> [Current Selection]:

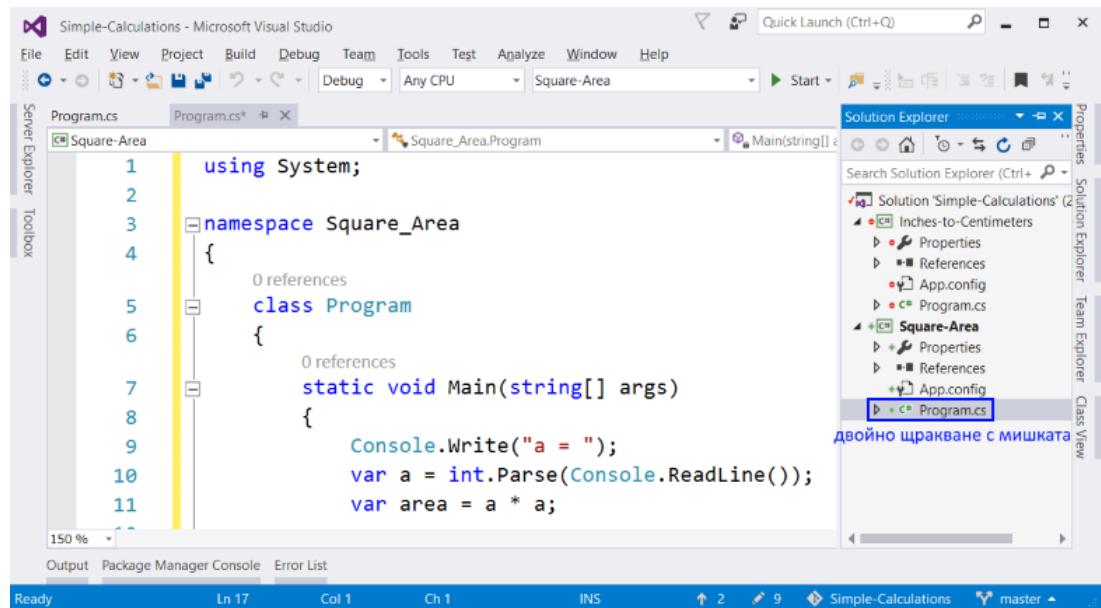




Сега отново **стартираме програмата**, както обикновено с [Ctrl+F5]. Този път ще се стартира **текущата отворена програма**, която преобразува инчове в сантиметри. Изглежда работи коректно:



Сега **да превключим** към преходната програма (лице на квадрат). Това става с двоен клик на мишката върху файла **Program.cs** от предходния проект “**Square-Area**” в панела [Solution Explorer] на Visual Studio:



Натискаме пак [Ctrl+F5]. Този път трябва да се стартира другият проект:

```
C:\WINDOWS\system32... - X
a = 3
Square area = 9
Press any key to continue . . .
```

Превключваме обратно към проекта "Inches-to-Centimeters" и го стартираме с [Ctrl + F5]:

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

Превключването между проектите е много лесно, нали? Просто избираме файла със сорс кода на програмата, кликваме го два пъти с мишката и при стартиране тръгва програмата от този файл.

Да тестваме с дробни числа, например с 2.5:

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



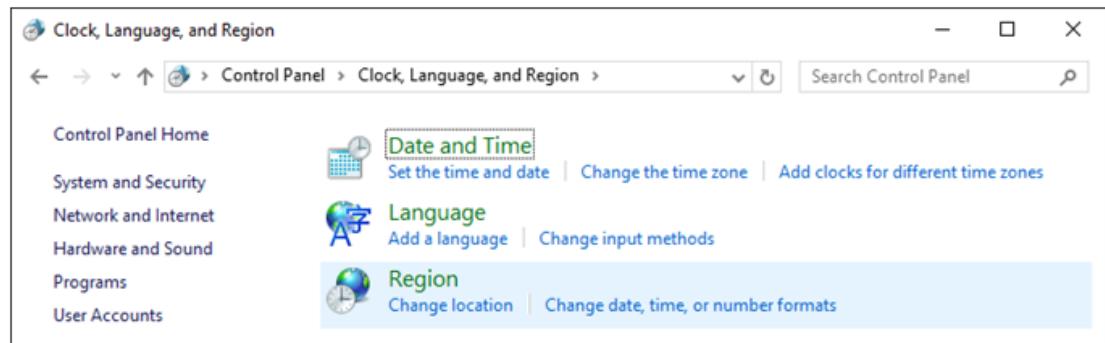
В зависимост от регионалните настройки на операционната система, е възможно вместо **десетична точка** (US настройки) да се използва **десетична запетая** (BG настройки).

Ако програмата очаква **десетична точка** и бъде въведено число с **десетична запетая** или обратното (бъде въведена десетична точка, когато се очаква десетична запетая), ще се получи следната грешка:

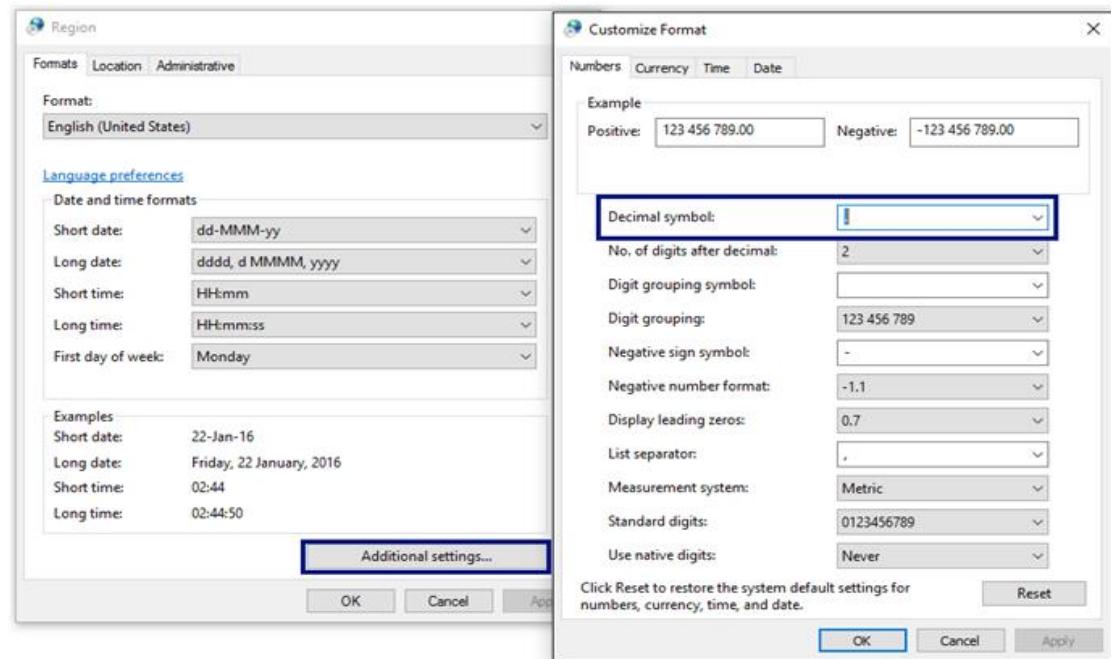
```
C:\WINDOWS\system32\cmd.exe - X
Inches = 2,5

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
=
```

Препоръчително е да променим настройките на компютъра си, така че да се използва десетична точка:



От настройките за регион в Windows (**Region**) изберете допълнителни настройки (**Additional settings...**) и в секцията за форматиране на числа задайте десетичният символ да е “.”, както е показано на картинката:



Тестване в Judge системата

Тествайте решението си: <https://judge.softuni.bg/Contests/Practice/Index/504#1>. Решението би трябвало да бъде прието като напълно коректно:

| Submissions | | |
|-------------|----------------------------------|---|
| Points | Time and memory used | Submission date |
| 100 / 100 | Memory: 7.83 MB Time: 0.014 s | 22:37:59 21.01.2016 Details |

Задача: поздрав по име

Да се напише програма, която чете от конзолата име на човек и отпечатва **Hello, <name>!**, където **<name>** е въведеното преди това име.

Насоки и подсказки

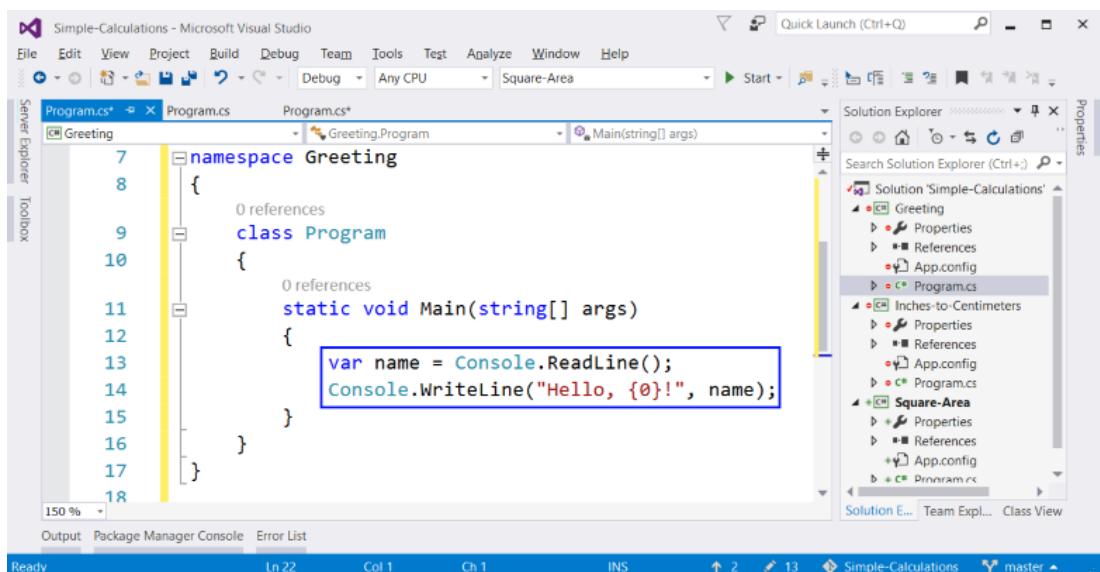
Първо създаваме нов C# конзолен проект с име “Greeting” в решението “Simple-Calculations”:

```

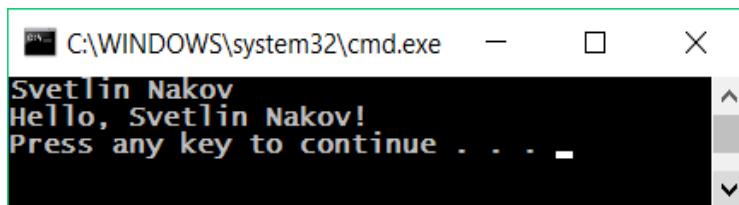
namespace Greeting
{
    class Program
    {
        static void Main(string[] args)
        {
        }
    }
}

```

Следва да напишем кода на програмата. Ако се затруднявате, може да ползвате примерния код по-долу:



Стартираме програмата с [Ctrl+F5] и я тестваме дали работи:



Тестване в Judge системата

Тествайте решението си: <https://judge.softuni.bg/Contests/Practice/Index/504#2>.

Задача: съединяване на текст и числа

Напишете C# програма, която прочита от конзолата име, фамилия, възраст и град и печата съобщение от следния вид: You are <firstName> <lastName>, a <age>-years old person from <town>.

Насоки и подсказки

Добавяме към текущото Visual Studio решение още един конзолен C# проект с име "Concatenate-Data". Пишем кода, който чете входните данни от конзолата:

```

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

```

Кодът, който отпечатва описаното в условието на задачата съобщение, трябва да се допише:



На горната картичка **кодът е нарочно даден замъглен**, за да помислите как да си го напишете сами. Ако даваме всичкия код, навсякъде наготово, мнозина ще го препишат механично и няма да се потрудят да си решат сами задачите. Затова в тази книга на много места ще даваме кода недописан, за **да пишете вие**.

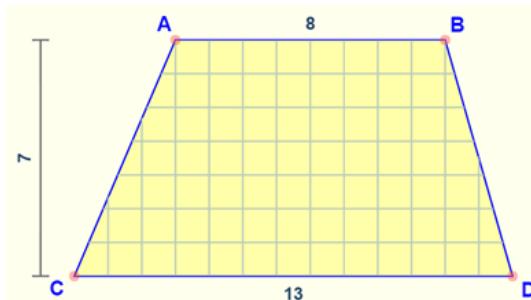
Тестване в Judge системата

Следва да се тества решението локално с [Ctrl+F5] и въвеждане на примерни входни данни. Тествайте решението си и в judge системата, която ще го провери автоматично: <https://judge.softuni.bg/Contests/Practice/Index/504#3>.

Задача: лице на трапец

Напишете програма, която чете от конзолата три числа b_1 , b_2 и h и пресмята лицето на трапец с основи b_1 и b_2 и височина h . Формулата за лице на трапец е $(b_1 + b_2) * h / 2$.

На фигурата по-долу е показан трапец със страни 8 и 13 и височина 7. Той има лице $(8 + 13) * 7 / 2 = 73.5$.



Насоки и подсказки

Отново трябва да добавим към текущото Visual Studio решение още един конзолен C# проект с име "Trapezoid-Area" и да напишем кода, който чете входните данни от конзолата, пресмята лицето на трапеца и го отпечатва:

```
static void Main(string[] args)
{
    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);
}
```

Кодът на картилката е нарочно размазан, за да помислите върху него и да го допишете сами.

Тествайте решението локално с [Ctrl+F5] и въвеждане на примерни данни.

Тестване в Judge системата

Тествайте решението си: <https://judge.softuni.bg/Contests/Practice/Index/504#4>.

Задача: периметър и лице на кръг

Напишете програма, която чете от конзолата **число r** и пресмята и отпечатва **лицето и периметъра на кръг/окръжност с радиус r**.

Примерен вход и изход

| Вход | Изход | Вход | Изход |
|------|---|------|---|
| 3 | Area = 28.2743338823081 Perimeter = 18.8495559215388 | 3 | Area = 28.2743338823081 Perimeter = 18.8495559215388 |

Насоки и подсказки

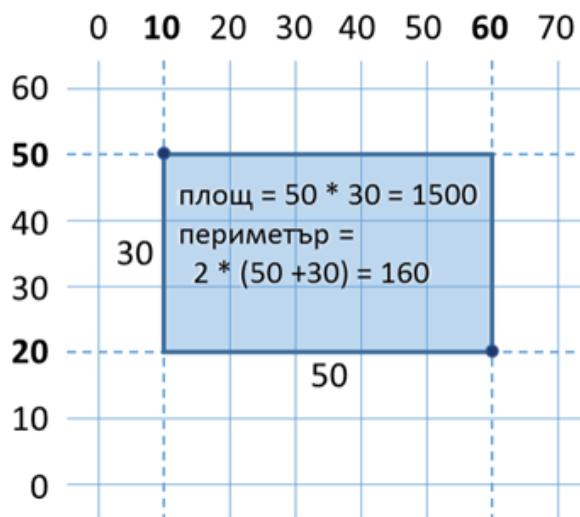
За изчисленията можете да използвате следните формули:

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

Тестване в Judge системата

Тествайте решението си: <https://judge.softuni.bg/Contests/Practice/Index/504#5>.

Задача: лице на правоъгълник в равнината



Правоъгълник е зададен с **координатите** на два от своите срещуположни ъгъла (x_1, y_1) – (x_2, y_2) .

Да се напише програма, която пресмята **площта и периметъра** му.

Входът се чете от конзолата. Числата x_1, y_1, x_2 и y_2 са дадени по едно на ред.

Изходът се извежда на конзолата и трябва да съдържа два реда с по една число на всеки от тях – лицето и периметъра.

Примерен вход и изход

| Вход | Изход |
|------|-------|
| 60 | |
| 20 | 1500 |
| 10 | 160 |
| 50 | |

| Вход | Изход |
|------|-------|
| 30 | |
| 40 | 2000 |
| 70 | 180 |
| -10 | |

| Вход | Изход |
|--------|-------------|
| 600.25 | 350449.6875 |
| 500.75 | 2402 |
| 100.50 | |
| -200.5 | |

Тестване в Judge системата

Тествайте решението си: <https://judge.softuni.bg/Contests/Practice/Index/504#6>.

Задача: лице на триъгълник

Напишете програма, която чете от конзолата **страна и височина на триъгълник** и пресмята неговото лице. Използвайте **формулата** за лице на триъгълник: $\text{area} = a * h / 2$. Закръглете резултата до **2 цифри** след десетичния знак (когато са налични). Може да използвате за закръглянето **Math.Round(area, 2)**.

Примерен вход и изход

| Вход | Изход |
|--------------|-----------------------|
| 20 30 | Triangle area = 300 |
| 7.75 8.45 | Triangle area = 32.74 |

| Вход | Изход |
|--------------------|-----------------------|
| 15 35 | Triangle area = 262.5 |
| 1.23456 4.56789 | Triangle area = 2.82 |

Тестване в Judge системата

Тествайте решението си: <https://judge.softuni.bg/Contests/Practice/Index/504#7>.

Задача: конзолен конвертор – от градуси °C към градуси °F

Напишете програма, която чете **градуси по скалата на Целзий (°C)** и ги преобразува до **градуси по скалата на Фаренхайт (°F)**. Потърсете в Интернет подходяща **формула**, с която да извършите изчисленията. Закръглете резултата до **2 символа** след десетичния знак. Ето няколко примера:

Примерен вход и изход

| Вход | Изход |
|------|-------|
| 25 | 77 |

| Вход | Изход |
|------|-------|
| 0 | 32 |

| Вход | Изход |
|------|-------|
| -5.5 | 22.1 |

| Вход | Изход |
|------|-------|
| 32.3 | 90.14 |

Тестване в Judge системата

Тествайте решението си: <https://judge.softuni.bg/Contests/Practice/Index/504#8>.

Задача: конзолен конвертор – от радиани в градуси

Напишете програма, която чете **ъгъл в радиани (rad)** и го преобразува в **градуси (deg)**. Потърсете в Интернет подходяща формула. Числото **π** в C# програмите е достъпно чрез **Math.PI**. Закръглете резултата до най-близкото цяло число използвайки метода **Math.Round(...)**.

Примерен вход и изход

| Вход | Изход |
|--------|-------|
| 3.1416 | 180 |
| 6.2832 | 360 |

| Вход | Изход |
|--------|-------|
| 0.7854 | 45 |
| 0.5236 | 30 |

Тестване в Judge системата

Тествайте решението си: <https://judge.softuni.bg/Contests/Practice/Index/504#9>.

Задача: конзолен конвертор – USD към BGN

Напишете програма за конвертиране на щатски долари (USD) в български лева (BGN). Закръглете резултата до **2 цифри** след десетичния знак. Използвайте фиксиран курс между долар и лев: **1 USD = 1.79549 BGN**.

Примерен вход и изход

| Вход | Изход |
|------|-----------|
| 20 | 35.91 BGN |

| Вход | Изход |
|------|------------|
| 100 | 179.55 BGN |

| Вход | Изход |
|------|-----------|
| 12.5 | 22.44 BGN |

Тестване в Judge системата

Тествайте решението си: <https://judge.softuni.bg/Contests/Practice/Index/504#10>.

Задача: * конзолен междувалутен конвертор

Напишете програма, която конвертира парична сума от една валута в друга. Трябва да се поддържат следните валути: BGN, USD, EUR, GBP. Използвайте следните фиксириани валутни курсове:

| Курс | USD | EUR | GBP |
|-------|---------|---------|---------|
| 1 BGN | 1.79549 | 1.95583 | 2.53405 |

Входът е **сума за конвертиране, входна валута и изходна валута**. Изходът е едно число – преобразуваната сума по посочените по-горе курсове, закръглен до **2 цифри** след десетичната точка.

Примерен вход и изход

| Вход | Изход |
|-------------------|-----------|
| 20 USD BGN | 35.91 BGN |
| 100 BGN EUR | 51.13 EUR |

| Вход | Изход |
|----------------------|------------|
| 12.35 EUR GBP | 9.53 GBP |
| 150.35 USD EUR | 138.02 EUR |

Тестване в Judge системата

Тествайте решението си: <https://judge.softuni.bg/Contests/Practice/Index/504#11>.

Задача: ** пресмятане с дати – 1000 дни на Земята

Напишете програма, която въвежда **рождена дата** във формат **dd-MM-yyyy** и пресмята датата, на която се навършват **1000 дни** от тази рождения дата и я отпечатва в същия формат.

Примерен вход и изход

| Вход | Изход |
|------------|------------|
| 25-02-1995 | 20-11-1997 |
| 07-11-2003 | 02-08-2006 |

| Вход | Изход |
|------------|------------|
| 14-06-1980 | 10-03-1983 |
| 01-01-2012 | 26-09-2014 |

| Вход | Изход |
|------------|------------|
| 30-12-2002 | 24-09-2005 |

Насоки и подсказки

- Потърсете информация за типа **DateTime** в C# и по-конкретно разгледайте **ParseExact(str, format)**, **AddDays(count)** и **ToString(format)**. С тяхна помощ може да решите задачата, без да е необходимо да изчислявате дни, месеци и високосни години.
- Не печатайте нищо допълнително на конзолата освен изискваната дата!

Тестване в Judge системата

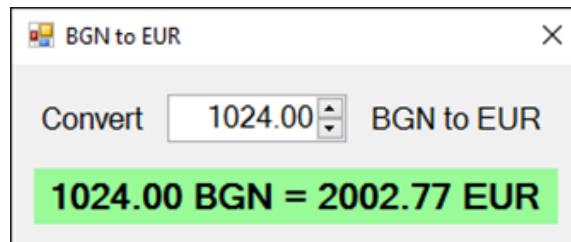
Тествайте решението си: <https://judge.softuni.bg/Contests/Practice/Index/504#12>.

Графични приложения с числови изрази

За да упражним работата с променливи и пресмятания с оператори и числови изрази, ще направим нещо интересно: ще разработим **настолно приложение** с **графичен потребителски интерфейс** (GUI desktop application). В него ще използваме пресмятания с дробни числа.

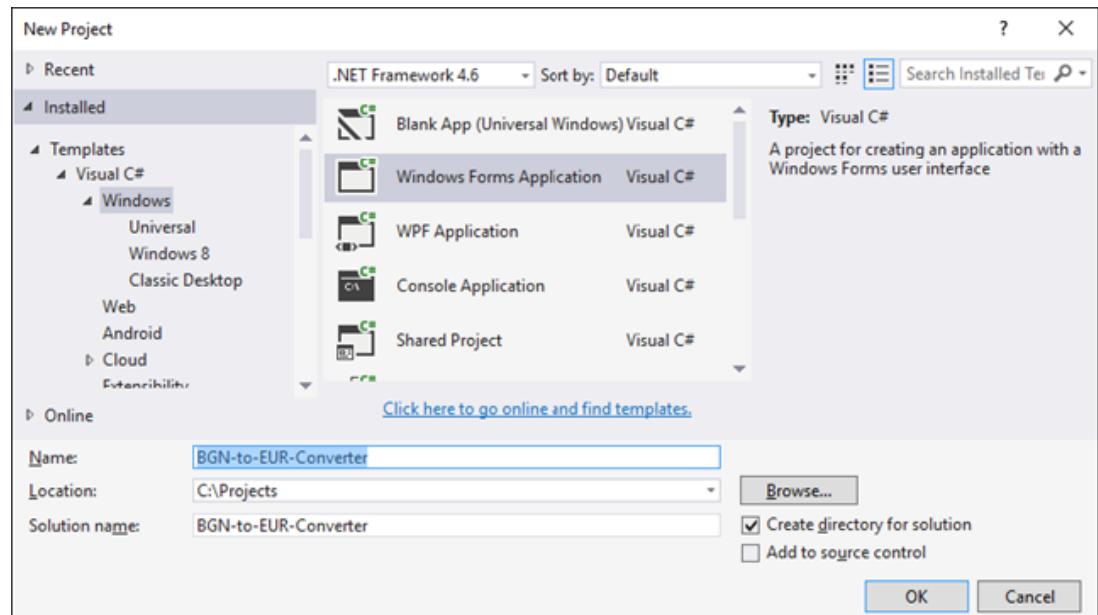
Графично приложение: конвертор от BGN към EUR

От нас се изиска да създадем **графично приложение** (GUI application), което пресмята стойността в **евро** (EUR) на парична сума, зададена в **лева** (BGN). При промяна на стойността в лева, равностойността в евро трябва да се преизчислява автоматично (използваме курс лева / евро: **1.95583**).



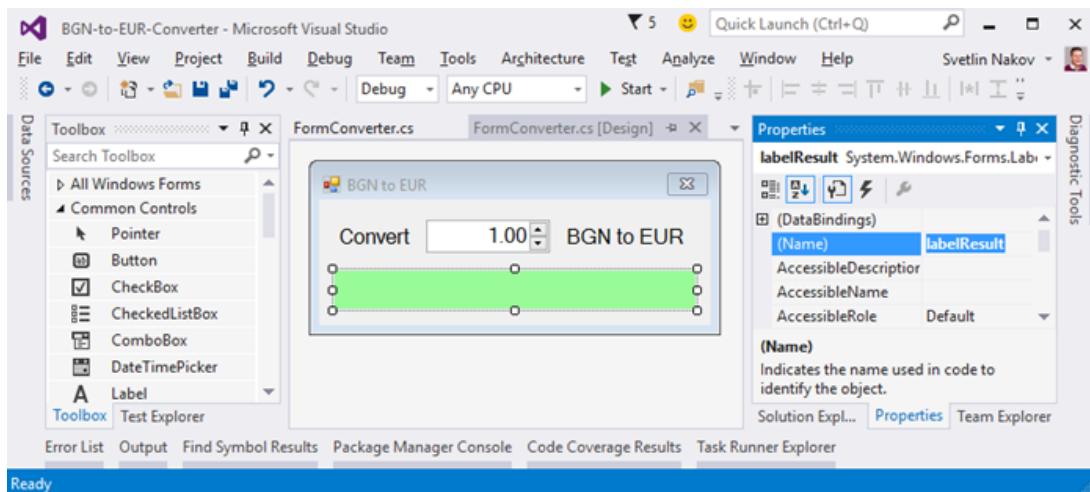
Тази задача излиза извън изучавания в книгата материал и има за цел не да ви научи как да програмирате GUI приложения, а **да ви запали интереса** към разработката на софтуер. Да се залавяме за работа.

Добавяме към текущото Visual Studio решение (solution) още един проект. Този път създаваме **Windows Forms** приложение със C# с име "BGN-to-EUR-Converter":



Подреждаме следните UI контроли във формата:

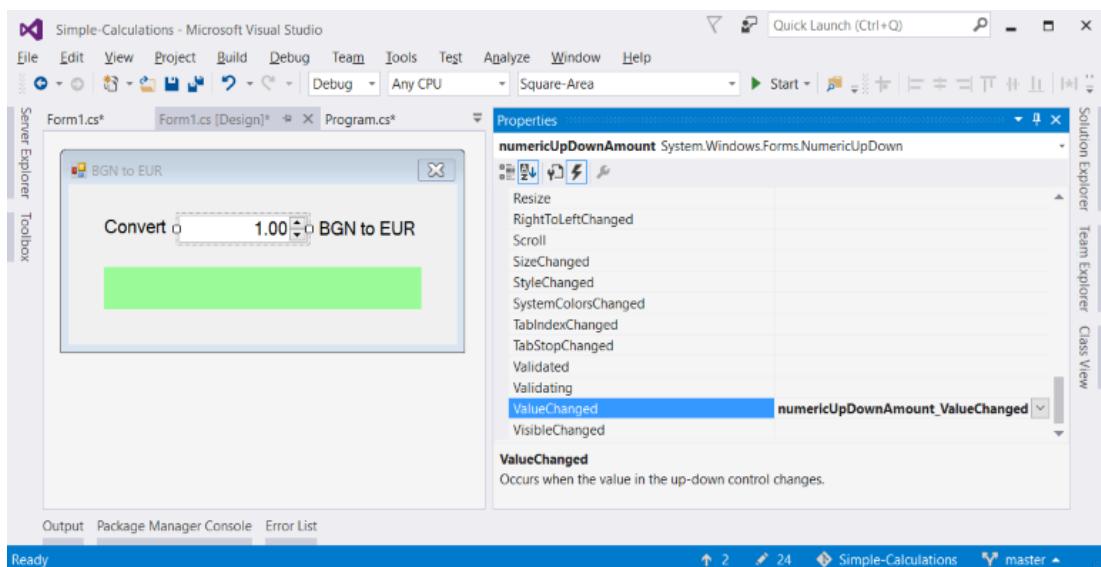
- **NumericUpDown** с име **numericUpDownAmount** – ще въвежда сумата за конвертиране
- **Label** с име **labelResult** – ще показва резултата след конвертиране
- Още два **Label** компонента, служещи единствено за статично изобразяване на текст Графичният редактор за потребителски интерфейс може да изглежда по подобен начин:



Задаваме следните настройки на формата и на отделните контроли:

| Настройка | Снимка |
|---|--------|
| FormConverter: Text = "BGN to EUR", Font.Size = 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, Font.Size = 14, Font.Bold = True | |

Дефинираме **обработчици на събития** по контролите. Това става като изберем дадена контрола, примерно полето за въвеждане на число, изберем някое от събитията в [Properties] прозореца, секция [Events] на Visual Studio и щракнем 2 пъти в дясно от него с мишката. На картинката по-долу е показано как да хванем събитието **ValueChanged** от контролата **NumericUpDown**:

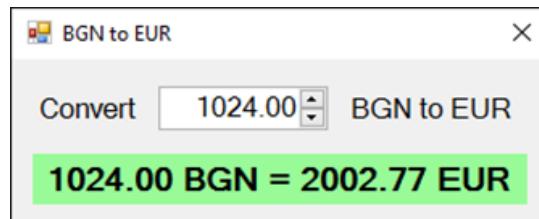


В нашия конвертор за валути ще хванем следните събития:

- **FormConverter.Load** (като кликнем върху формата 2 пъти с мишката)
- **numericUpDownAmount.ValueChanged** (като кликнем върху **NumericUpDown** контролата 2 пъти)
- **numericUpDownAmount.KeyUp** (избираме **Events** от таблото **Properties** и кликнем 2 пъти върху **KeyUp**)

Събитието **Form.Load** се изпълнява при стартиране на програмата, преди да се появи прозореца на приложението. Събитието **NumericUpDown.ValueChanged** се изпълнява при промяна на стойността в полето за въвеждане на число. Събитието **NumericUpDown.KeyUp** се изпълнява след натискане на клавиши в полето за въвеждане на число. При всяко от тези събития ще преизчисляваме резултата.

За хващане на събитие ползваме иконката със събитията (Events) в Properties прозореца във Visual Studio:



Ще използваме следния C# код за обработка на събитията:

```
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();
}

```

Всички прихванати събития извикват метода **ConvertCurrency()**, който конвертира зададената сума от лева в евро и показва резултата в зелената кутийка.

Трябва да напишем **кода** (програмната логика) за конвертиране от лева към евро:

```

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

```

Накрая **стартираме проекта** с [Ctrl+F5] и тестваме дали работи коректно.

Ако имате проблеми с примера по-горе, **гледайте видеото** в началото на тази глава. Там приложението е направено на живо стъпка по стъпка с много обяснения. Или питайте във **форума на СофтУни**: <https://softuni.bg/forum>.

Графично приложение: *** Хвани бутона!

Създайте забавно графично приложение „**хвани бутона**“: една форма съдържа един бутоン. При преместване на курсора на мишката върху бутона той се премества на случайна позиция. Така се създава усещане, че „**бутонът бяга от мишката и е трудно да се хване**“.

При „хващане“ на бутона се извежда съобщение-поздрав.

Примерна визуализация на приложението е достъпна на картинката по-долу:



Подсказка: напишете обработчик за събитието **Button.MouseEnter** и премествайте бутона на случайна позиция. Използвайте генератор за случайни числа **Random**. Позицията на бутона се задава от свойството **Location**. За да бъде новата позиция на бутона в рамките на формата, можете да направите изчисления спрямо размера на формата, който е достъпен от свойството **ClientSize**. Можете да ползвате следния примерен код:

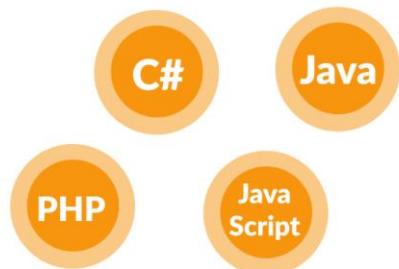
```
private void buttonCatchMe_MouseEnter(
    object sender, EventArgs e)
{
    Random rand = new Random();
    var maxWidth =
        this.ClientSize.Width - buttonCatchMe.ClientSize.Width;
    var maxHeight =
        this.ClientSize.Height - buttonCatchMe.ClientSize.Height;
    this.buttonCatchMe.Location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

Ако имате трудности, можете да потърсите помощ във форума на СофтУни: <https://softuni.bg/forum>.

Качествено образование,
професия и работа за

Софтуерни инженери

- ✓ Безплатен старт за начинаещи - **присъствено и онлайн**
- ✓ Избор измежду **най-търсените професии** в софтуерната индустрия
- ✓ Съдействие за **кариерен старт**
- ✓ Отлични преподаватели, ментори и студентска общност



Софтууни предоставя съвременно иновативно образование за програмиране, ИТ и дигитални умения на хиляди млади хора. Програмата "**Софтуерен университет**" изгражда истински **професионалисти в света на програмирането**.

Включете се в цялостната програма по софтуерно инженерство на Софтууни, за да усвоите **най-търсените съвременни софтуерни технологии** чрез модерна учебна методика и с много практически задачи и проекти. **Учебният план** е разработен съвместно с ИТ фирмите и изгражда най-търсените от тях умения.

Софтууни работи пряко с компаниите от софтуерната индустрия, съдейтайки на своите студенти за реализацията им като успешни софтуерни инженери.

ПЪТЯТ НА СТУДЕНТА В СОФТУНИ



Programming Basics

Кариерен Старт

Дипломиране

70/100 credits

80/100/150 credits

Кандидатствай

softuni.bg/apply

Глава 2.2. Прости пресмятания с числа

– ИЗПИТНИ задачи

В предходната глава се запознахме със системната конзола и как да работим с нея – как да прочетем число от конзолата и как да отпечатаме резултат на конзолата. Разглеждахме основните аритметични операции и накратко споменахме типовете данни. В настоящата глава ще упражним и затвърдим наученото досега, като разгледаме няколко по-сложни задачи, давани на изпити в СофтУни.

Четене на числа от конзолата

Преди да преминем към задачите, да си припомним най-важното от изучавания материал от предходната тема. Ще започнем с четенето на числа от конзолата.

Четене на цяло число

Необходима ни е променлива, в която да запазим числото (напр. `num`), и да използваме стандартната команда за четене на данни от конзолата в съчетание с функцията `int.Parse(...)`, която конвертира текст в число:

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

Четене на дробно число

По същия начин, както четем цяло число, но този път ще използваме функцията `double.Parse(...)`:

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

Извеждане на текст по шаблон (placeholder)

Placeholder представлява израз, който ще бъде заменен с конкретна стойност при отпечатване. Методите `Console.WriteLine()` / `WriteLine(...)` поддържат печатане на текст по шаблон, като първият аргумент, който трябва да подадем, е форматиращият низ, следван от броя аргументи, равен на броя на плейсхолдърите.

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

Аритметични оператори

Да си припомним основните аритметични оператори за пресмятания с числа.

Оператор +

```
var result = 3 + 5; // резултатът е 8
```

Оператор -

```
var result = 3 - 5; // резултатът е -2
```

Оператор *

```
var result = 3 * 5; // резултатът е 15
```

Оператор /

```
var result = 7 / 3; // резултатът е 2 (целочислено деление)
var result2 = 5 / 2.0; // резултатът е 2.5 (дробно деление)
```

Конкатенация

При използване на оператора **+** между променливи от тип текст (или между текст и число) се извършва т.нр. конкатенация (сплеване на низове).

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

Изпитни задачи

Сега, след като си припомнихме как се извършват пресмятания с числа и как се четат и печатат числа на конзолата, да минем към задачите. Ще решим няколко задачи от приемен изпит за кандидатстване в СофтУни.

Задача: учебна зала

Учебна зала има правоъгълен размер **l** на **w** метра, без колони във вътрешността си. Залата е разделена на две части – лява и дясна, с коридор – приблизително по средата. В лявата и в дясната част има **редици с бюра**. В задната част на залата има голяма **входна врата**. В предната част на залата има **катедра** с подиум за преподавателя. Едно **работно място** заема **70 на 120 cm** (маса с размер 70 на 40 см + място за стол и преминаване с размер 70 на 80 см). **Коридорът** е широк поне **100 см**. Изчислено е, че заради **входната врата** (която е с отвор 160 см) се губи точно **1 работно място**, а заради **катедрата** (която е с размер 160 на 120 см) се губят **2**

работни места. Напишете програма, която въвежда размери на учебната зала и изчислява **броя работни места в нея** при описаното разположение (вж. фигурата).

Входни данни

От конзолата се четат **2 числа**, по едно на ред: l (дължина в метри) и w (широкина в метри). Ограничения: $3 \leq w \leq l \leq 100$.

Изходни данни

Да се отпечата на конзолата едно цяло число: **броят места** в учебната зала.

Примерен вход и изход

| Вход | Изход | Чертеж |
|------------|-------|--|
| 15 8.9 | 129 |  <p>коридор: поне 1 m</p> <p>врата</p> <p>катедра</p> |
| 8.4 5.2 | 39 |  <p>коридор: 1 m</p> <p>врата</p> <p>катедра</p> |

Пояснения към примерите

В първия пример залата е дълга **1500** см. В нея могат да бъдат разположени **12 реда** ($12 * 120 \text{ cm} = 1440 + 60 \text{ см остатък}$). Залата е широка **890** см. От тях 100 см отиват за коридора в средата. В останалите 790 см могат да се разположат по **11 бюра на ред** ($11 * 70 \text{ cm} = 770 \text{ cm} + 20 \text{ см остатък}$). **Брой места = $12 * 11 - 3 = 132 - 3 = 129$** (имаме 12 реда по 11 места = 132 минус 3 места за катедра и входна врата).

Във втория пример залата е дълга **840** см. В нея могат да бъдат разположени **7 реда** ($7 * 120 \text{ cm} = 840$, без остатък). Залата е широка **520** см. От тях 100 см отиват за коридора в средата. В останалите 420 см могат да се разположат по **6 бюра на ред** ($6 * 70 \text{ cm} = 420 \text{ см}$, без остатък). **Брой места = $7 * 6 - 3 = 42 - 3 = 39$** (имаме 7 реда по 6 места = 42 минус 3 места за катедра и входна врата).

Насоки и подсказки

Опитайте първо сами да решите задачата. Ако не успеете, разгледайте насоките и подсказките.

Идея за решение

Както при всяка една задача по програмиране, е **важно да си изградим идея за решението ѝ**, преди да започнем да пишем код. Да разгледаме внимателно зададеното ни условие. Изиска се да напишем програма, която да изчислява броя работни места в една зала, като този брой е зависим от дълчината и височината ѝ. Забелязваме, че те ще ни бъдат подадени като входни данни **в метри**, а информацията за това колко пространство заемат работните места и коридорът, ни е дадена **в сантиметри**. За да извършим изчисленията, ще трябва да използваме еднакви мерни единици, няма значение дали ще изберем да превърнем височината и дълчината в сантиметри, или останалите данни в метри. За представеното тук решение е избрана първата опция.

Следва да изчислим **колко колони и колко редици** с бюра ще се съберат. Колоните можем да пресметнем като от широчината **извадим необходимото място за коридора (100 см)** и **разделим остатъка на 70 см** (колкото е дълчината на едно работно място). Редиците ще намерим като разделим **дълчината на 120 см**. И при двете операции може да се получи **реално число** с цяла и дробна част, **в променлива трябва да запазим обаче само цялата част**. Накрая умножаваме броя на редиците по този на колоните и от него изваждаме 3 (местата, които се губят заради входната врата и катедрата). Така ще получим исканата стойност.

Избор на типове данни

От примерните входни данни виждаме, че за вход може да ни бъде подадено реално число с цяла и дробна част, затова не е подходящо да избираме тип **int**, нека за тях използваме **double**. Изборът на тип за следващите променливи зависи от метода за решение, който изберем. Както всяка задача по програмиране, тази също има **повече от един начин на решение**. Тук ще бъдат показани два такива.

Решение

Време е вече да пристъпим към решението. Мислено можем да го разделим на три подзадачи:

- Прочитане на **входните данни**.
- Извършване на **изчисленията**.
- Извеждане на **изход** на конзолата.

Първото, което трябва да направим, е да прочетем входните данни от конзолата.

С **Console.ReadLine()** четем стойностите от конзолата, а с функцията **double.Parse(...)** преобразуваме зададената стрингова (текстова) стойност в **double**.

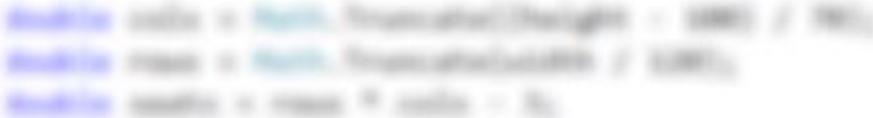
```
double length = double.Parse(Console.ReadLine());
double width = double.Parse(Console.ReadLine());
```

Нека пристъпим към изчисленията. Особеното тук е, че след като извършим делението, трябва да запазим в променлива само **цялата част от резултата**.



Търсете в Google! Винаги, когато имаме идея как да решим даден проблем, но не знаем как да го имплементираме на C#, или когато се сблъскаме с такъв, за който предполагаме, че много други хора са имали, най-лесно е да се справим като потърсим информация в Интернет.

В случая пробваме със следното търсене: "*c# get whole number part of double*". Откриваме, че едната възможност е да използваме метода **Math.Truncate(...)**. Тъй като той работи с променливи от тип **double**, за броя редици и колони създаваме променливи също от този тип.



Втори вариант: както вече знаем, операторът за деление **/** има различно действие върху цели и реални числа. **При деление на целочислен с целочислен тип** (напр. **int**), **върнатият резултат е отново целочислен**. Следователно можем да потърсим как да преобразуваме реалните числа, които имаме като стойности за височината и широчината, в цели числа и след това да извършим делението.

Тъй като в този случай може да се получи **загуба на данни**, идваща от премахването на дробната част, е необходимо преобразуването да стане **изрично** (explicit typecasting). Използваме оператора за преобразуване на данни (**type**), като заменяме думата **type** с необходимия **тип данни** и го поставяме **преди** променливата.

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

С **Console.WriteLine(...)** отпечатваме резултата на конзолата.

Тестване в Judge системата

Тествайте решението си: <https://judge.softuni.bg/Contests/Practice/Index/505#0>.

Задача: зеленчукова борса

Градинар продава реколтата от градината си на зеленчукова борса. Продава зеленчуци за N лева на килограм и плодове за M лева за килограм. Напишете програма, която да пресмята приходите от реколтата в евро, ако се приеме, че **едно евро** е равно на **1.94** лв.

Входни данни

От конзолата се четат **4 числа**, по едно на ред:

- Първи ред – цена за килограм зеленчуци – число с плаваща запетая.
- Втори ред – цена за килограм плодове – число с плаваща запетая.
- Трети ред – общо килограми на зеленчуците – цяло число.
- Четвърти ред – общо килограми на плодовете – цяло число.

Ограничения: всички числа ще са в интервала от **0.00** до **1000.00**.

Изходни данни

Да се отпечата на конзолата **едно число с плаваща запетая**: приходите от всички плодове и зеленчуци в евро.

Примерен вход и изход

| Вход | Изход | Вход | Изход |
|-------|-------|------|------------------|
| 0.194 | | 1.5 | |
| 19.4 | 101 | 2.5 | |
| 10 | | 10 | 20.6185567010309 |
| 10 | | 10 | |

Пояснения към първия пример:

- Зеленчуците струват: 0.194 лв. * 10 кг. = **1.94** лв.
- Плодовете струват: 19.4 лв. * 10 кг. = **194** лв.
- Общо: 195.94 лв. = **101** евро.

Насоки и подсказки

Първо ще дадем няколко разсъждения, а след това и конкретни насоки за решаване на задачата, както и съществената част от кода.

Идея за решение

Нека първо разгледаме зададеното ни условие. В случая, от нас се иска да пресметнем колко е общият приход от реколтата. Той е равен на **сбора от печалбата от плодовете и зеленчуците**, а тях можем да изчислим като умножим **цената на килограм по количеството им**. Входните данни са дадени в лева, а за изхода се изисква да бъде в евро. По условие 1 евро е равно на 1.94 лева, следователно за да получим исканата **изходна стойност**, трябва да разделим **сбора** на **1.94**.

Избор на типове данни

След като сме изяснили идеята си за решаването на задачата, можем да пристъпим към избора на подходящи типове данни. Да разгледаме **входа**: дадени са **две цели числа** за общия брой килограми на зеленчуците и плодовете, съответно променливите, които декларираме, за да пазим техните стойности, ще бъдат от тип **int**. За цените на плодовете и зеленчуците е указано, че ще бъдат подадени **две числа с плаваща запетая**, т.е. променливите ще бъдат от тип **double**.

Може да декларираме също две променливи, в които да пазим стойността на печалбата от плодовете и зеленчуците поотделно. Тъй като умножаваме променлива от тип **int** (общо килограми) с такава от тип **double** (цена), резултатът също трябва да бъде от тип **double**. Нека поясним това: по принцип **операторите работят с аргументи от един и същи тип**. Следователно, за да извършим операция като умножение върху два различни типа данни, ни се налага да ги преобразуваме към един и същ такъв. Когато в един израз има типове с различен обхват, преобразуването винаги се извършва към този с най-голям обхват, в този случай това е **double**. Тъй като няма опасност от загуба на данни, **преобразуването е неявно** (**implicit**) и става автоматично от компилатора.

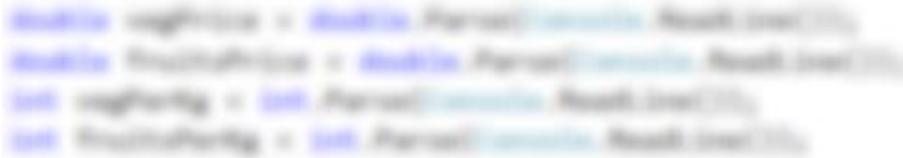
Като **изход** се изисква също **число с плаваща запетая**, т.е. резултата ще пазим в променлива от тип **double**.

Решение

Време е да пристъпим към решението. Мислено можем да го разделим на три подзадачи:

- Прочитане на **входните данни**.
- Извършване на **изчисленията**.
- Извеждане на **изход** на конзолата.

За да прочетем входните данни декларираме променливи, като внимаваме да ги именуваме по такъв начин, който да ни подсказва какви стойности съдържат променливите. Чрез **Console.ReadLine(...)** четем стойностите от конзолата, а с функциите **int.Parse(...)** и **double.Parse(...)** преобразуваме зададената стрингова стойност съответно в **int** и **double**.



Извършваме необходимите изчисления:

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

В условието на задачата не е зададено специално форматиране на изхода, следователно трябва просто да изчислим исканата стойност и да я отпечатаме на конзолата. Както в математиката, така и в програмирането делението има приоритет пред събирането. За задачата обаче трябва първо да **сметнем** **сбора** на двете получени стойности и след това да **разделим на 1.94**. За да дадем предимство на събирането, може да използваме скоби. С **Console.WriteLine(...)** отпечатваме изхода на конзолата.

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

Тестване в Judge системата

Тествайте решението си: <https://judge.softuni.bg/Contests/Practice/Index/505#1>.

Задача: ремонт на плочки

На площадката пред жилищен блок трябва да се поставят плочки. Площадката е с форма на квадрат със страна **N** метра. Плочките са широки „**W**“ метра и дълги „**L**“ метра. На площадката има една пейка с **ширина M** метра и **дължина O** метра. Под нея не е нужно да се слагат плочки. Всяка плочка се поставя за **0.2 минути**.

Напишете програма, която чете от конзолата размерите на **площадката**, **плочките** и **пейката** и пресмята **колко плочки са необходими** да се покрие площадката и пресмята времето за поставяне на всички плочки.

Пример: площадка с размер 20 м. има площ 400 кв. м. Пейка, широка 1 м. и дълга 2 м., заема площ 2 кв. м. Една плочка е широка 5 м. и дълга 4 м. и има площ = 20 кв. м. Площта, която трябва да се покрие, е $400 - 2 = 398$ кв. м. Необходими са $398 / 20 = 19.90$ плочки. Необходимото време е $19.90 * 0.2 = 3.98$ минути.

Входни данни

От конзолата се четат **5 числа**:

- **N** – дължината на **страна** от **площадката** в интервала [1 ... 100].
- **W** – широчината на една **плочка** в интервала [0.1 ... 10.00].
- **L** – дължината на една **плочка** в интервала [0.1 ... 10.00].
- **M** – широчината на **пейката** в интервала [0 ... 10].

- О – дължината на пейката в интервала [0 ... 10].

Изходни данни

Да се отпечатат на конзолата **две числа**: броя **плочки**, необходим за ремонта и времето за поставяне, всяко на нов ред.

Примерен вход и изход

| Вход | Изход | Вход | Изход |
|------|-------|------|-----------------|
| 20 | | 40 | |
| 5 | 19.9 | 0.8 | 3302.0833333333 |
| 4 | 3.98 | 0.6 | 660.41666666667 |
| 1 | | 3 | |
| 2 | | 5 | |

Обяснения за първия пример:

- Обща площ = $20 * 20 = 400$.
- Площ на пейката = $1 * 2 = 2$.
- Площ за покриване = $400 - 2 = 398$.
- Площ на плочки = $5 * 4 = 20$.
- Необходими плочки = $398 / 20 = 19.9$.
- Необходимо време = $19.9 * 0.2 = 3.98$.

Насоки и подсказки

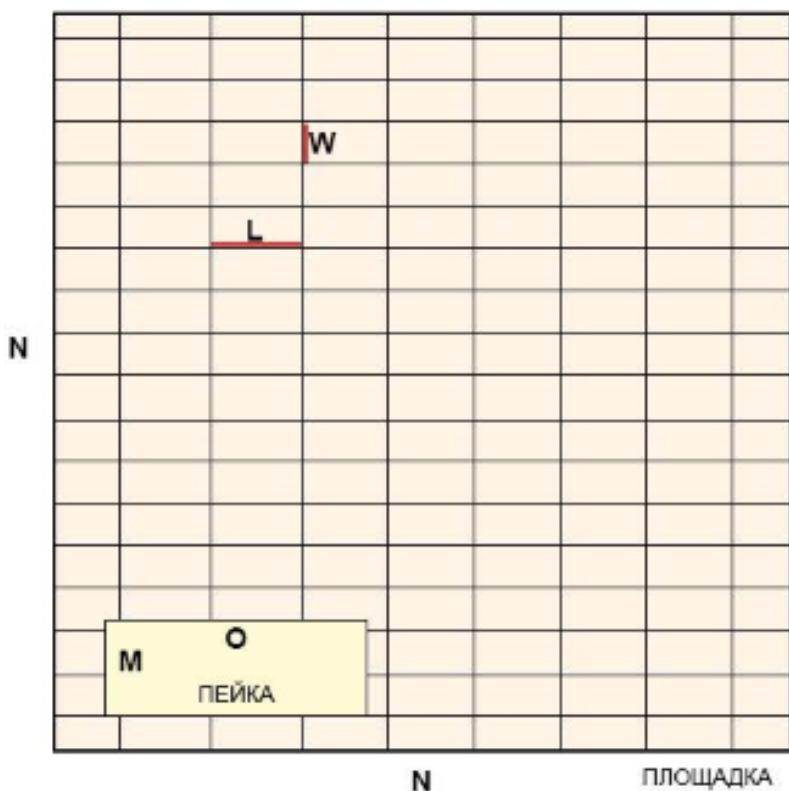
Нека **да си направим чертеж**, за да поясним условието на задачата. Той може да изглежда като на фигурата по-долу. По чертежа можем да разсъждаваме много по-лесно, използвайки визуална представя за ситуацията от задачата.

Идея за решение

Изиска се да пресметнем **броя плочки**, който трябва да се постави, както и времето, за което това ще се извърши. За да **изчислим броя**, е необходимо да сметнем **площта**, която трябва да се покрие, и да я **разделим на лицето на една плочка**. По условие площадката е квадратна, следователно общата площ ще намерим като умножим страната ѝ по стойността **W * L**. След това пресмятаме **площта, която заема пейката**, също като умножим двете ѝ страни **M * O**. Като извадим площта на пейката от тази на цялата площадка, получаваме площта, която трябва да се ремонтира.

Лицето на единична плоча изчисляваме като **умножим едната ѝ страна по другата W * L**. Както вече отбелязахме, сега трябва да **разделим площта за покриване на площта на една плочка**. По този начин ще разберем какъв е необходимият брой

плочки. Него умножаваме по **0.2** (времето, за което по условие се поставя една плочка). Така вече ще имаме исканите изходни стойности.



Избор на типове данни

Дължината на страна от площадката, широчината и дължината на пейката ще бъдат дадени като **цели числа**, следователно, за да запазим техните стойности може да декларираме **променливи от тип int**. За широчината и дължината на плочките ще ни бъдат подадени реални числа (с цяла и дробна част), затова за тях ще използваме **double**. Изходът на задачата отново ще е реално число, т.е. променливите ще бъдат също от тип **double**.

Решение

Както и в предходните задачи, можем мислено да разделим решението на три части:

- Прочитане на **входните данни**.
- Извършване на **изчисленията**.
- Извеждане на **изход** на конзолата.

Първото, което трябва да направим, е да прочетем правилно **входните данни** на задачата. Важно е да внимаваме за последователността, в която са дадени. Чрез **Console.ReadLine(...)** четем стойностите от конзолата, а с **int.Parse(...)** и

`double.Parse(...)` преобразуваме зададената стрингова стойност, съответно в `int` и `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());
```

След като сме инициализирали променливите и сме запазили съответните стойности в тях, пристъпваме към **изчисленията**. Тъй като стойностите на променливите `n`, `a` и `b`, с които работим, са запазени в променливи от тип `int`, за резултатите от изчисленията може да дефинираме **променливи също от този тип**.

Променливите `w` и `h` са от тип `double`, т.е. за **лицето на една плочка** създаваме променлива от същия тип. За финал **изчисляваме** стойностите, които трябва да **отпечатаме** на конзолата. **Броят** на необходимите **плочки** получаваме като **разделим** **площта**, която трябва да се покрие, на **площта на единична плочка**. При деление на две числа, от които **едното е реално**, резултатът е **реално число** с цяла и дробна част. Следователно, за да са коректни изчисленията ни, запазваме резултата в променлива от тип `double`. В условието на задачата не е зададено специално форматиране или закръгляне на изхода, затова просто отпечатваме стойностите с `Console.WriteLine(...)`.

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

double time = tiles * 0.2;

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

Тестване в Judge системата

Тествайте решението си: <https://judge.softuni.bg/Contests/Practice/Index/505#2>.

Задача: парички

Преди време **Пешо си е купил биткойни**. Сега ще ходи на екскурзия из Европа и ще му трябва евро. Освен биткойни има и **китайски юани**. Пешо иска да обмени парите си в евро за екскурзиията. Напишете програма, която да пресмята колко евро може да купи спрямо следните валутни курсове:

- 1 биткойн = 1168 лева.
- 1 китайски юан = 0.15 долара.
- 1 доллар = 1.76 лева.
- 1 евро = 1.95 лева.

Обменното бюро има **комисионна** от **0%** до **5%** от крайната сума в **евро**.

Входни данни

От конзолата се четат 3 числа:

- На първия ред – **броят биткойни**. Цяло число в интервала [0 ... 20].
- На втория ред – **броят китайски юани**. Реално число в интервала [0.00 ... 50 000.00].
- На третия ред – **комисионната**. Реално число в интервала [0.00 ... 5.00].

Изходни данни

На конзолата да се отпечата едно число – **резултатът от обмяната на валутите**. Не е нужно резултатът да се закръгля.

Примерен вход и изход

| Вход | Изход |
|------|------------------|
| 1 | |
| 5 | |
| 5 | 569.668717948718 |

| Вход | Изход |
|------|------------------|
| 20 | |
| 5678 | |
| 2.4 | 12442.2442010256 |

| Вход | Изход |
|----------|------------------|
| 7 | |
| 50200.12 | |
| 3 | 10659.4701177436 |

Обяснение за първия пример:

- 1 биткойн = 1168 лева
- 5 юана = 0.75 долара
- 0.75 долара = 1.32 лева

- $1168 + 1.32 = 1169.32$ лева = 599.651282051282 евро
- Комисионна: 5% от 599.651282051282 = 29.9825641025641
- Резултат: $599.651282051282 - 29.9825641025641 = 569.668717948718$ евро

Насоки и подсказки

Нека отново помислим първо за начина, по който можем да решим задачата, преди да започнем да пишем код.

Идея за решение

Виждаме, че ще ни бъдат подадени броят биткойни и броят китайски юани. За **изходната стойност** е указано да бъде в **евро**. В условието са посочени и валутните курсове, с които трябва да работим. Забелязваме, че към евро можем да преобразуваме само сума в лева, следователно трябва **първо да пресметнем цялата сума**, която Пешо притежава, **в лева**, и **след това да изчислим изходната стойност**.

Тъй като ни е дадена информация за валутния курс на биткойни срещу лева, можем директно да направим това преобразуване. От друга страна, за да получим стойността на **китайските юани в лева**, трябва първо да ги конвертираме в долари, а след това **доларите – в лева**. Накрая ще **съберем двете получени стойности** и ще пресметнем на колко евро съответстват.

Остава последната стъпка: да **пресметнем колко ще бъде комисионната** и да извадим получената сума от общата. Като комисионна ще ни бъде подадено **реално число**, което ще представлява определен **процент от общата сума**. Нека още в началото разделим подаденото число на 100, за да изчислим **процентната му стойност**. Няя ще умножим по сумата в евро, а резултатът ще извадим от същата тази сума. Получената сума ще отпечатаме на конзолата.

Избор на типове данни

Биткойните са дадени като **цяло число**, следователно за тяхната стойност може да декларираме **променлива от тип int**. Като брой **китайски юани и комисионна** ще получим **реално число**, следователно за тях използваме **double**. Тъй като типът данни **double** е с по-голям обхват, а **изходът** също ще бъде **реално число**, ще използваме него и за останалите променливи, които създаваме.

Решение

След като сме си изградили идея за решението на задачата и сме избрали структурите от данни, с които ще работим, е време да пристъпим към **писането на код**. Както и в предните задачи, можем да разделим решението на три подзадачи:

- Прочитане на **входните данни**.
- Извършване на **изчисленията**.
- Извеждане на **изход** на конзолата.

Декларираме променливите, които ще използваме, като отново внимаваме да изберем **смислени имена**, които подсказват какво съдържат те. Инициализираме техните стойности: с **Console.ReadLine(...)** четем подадените числа на конзолата и конвертираме въведения от потребителя стринг към **int** или **double**.

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

Извършваме необходимите изчисления:

```
decimal bitcoinsToLeva = bitcoins * 1168;
decimal yuansToDollars = yuans * 0.15m;
decimal dollarsToLeva = yuansToDollars * 1.76m;
```

Накрая пресмятаме стойността на комисионната и я изваждаме от сумата в евро. Нека обърнем внимание на начина, по който можем да изпишем това: **euro -= commission * euro** е съкратен начин за изписване на **euro = euro - (commission * euro)**. В случая използваме комбиниран оператор за присвояване (**-=**), който изважда стойността от операнда **вдясно от този вляво**. Операторът за умножение (*****) има **по-висок приоритет** от комбинирирания оператор, затова изразът **commission * euro** се изпълнява първи, след което неговата стойност се изважда.

В условието на задачата не е зададено специално форматиране или закръгляне на резултата, следователно трябва просто да изчислим изхода и да го отпечатаме на конзолата без допълнителна обработка.

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

Нека обърнем внимание на нещо, което важи за всички задачи от този тип: разписано по този начин, решението на задачата е доста подробно. Тъй като условието като цяло не е сложно, бихме могли на теория да напишем един **дълъг израз**, в който директно след получаване на входните данни да сметнем изходната стойност. Например, такъв израз би изглеждал ето така:

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

Този код би дал правилен резултат, но се чете трудно. Няма да ни е лесно да разберем какво прави и дали съдържа грешки, както и как да поправим някоя такава. По-добра практика е **вместо един сложен израз да напишем няколко прости** и да запишим резултатите от тях в променливи със подходящи имена. Така кодът е ясен и по-лесно променяем.

Тестване в Judge системата

Тествайте решението си: <https://judge.softuni.bg/Contests/Practice/Index/505#3>.

Задача: дневна печалба

Иван е програмист в американска компания и работи от вкъщи средно N дни в месеца, като изкарва средно по M долара на ден. В края на годината Иван получава бонус, който е равен на 2.5 месечни заплати. От спечеленото през годината му се удържат 25% данъци. Напишете програма, която да пресмята колко е чистата средна печалба на Иван на ден в лева, тъй като той харчи изкараното в България. Приема се, че в годината има точно 365 дни. Курсът на долара спрямо лева ще се чете от конзолата.

Входни данни

От конзолата се четат 3 числа:

- На първия ред – **работни дни в месеца**. Цяло число в интервала [5 ... 30].
- На втория ред – **изкарани пари на ден**. Реално число в интервала [10.00 ... 2000.00].
- На третия ред – **курсът на долара спрямо лева** (1 доллар = X лева). Реално число в интервала [0.99 ... 1.99].

Изходни данни

На конзолата да се отпечата 1 число – средната печалба на ден в лева. Резултатът да се форматира до втората цифра след десетичния знак.

Примерен вход и изход

| Вход | Изход | Вход | Изход | Вход | Изход |
|-------|-------|------|-------|--------|--------|
| 21 | | 15 | | 22 | |
| 75.00 | 74.61 | 105 | 80.24 | 199.99 | 196.63 |
| 1.59 | | 1.71 | | 1.50 | |

Обяснение за първия пример:

- 1 месечна заплата = $21 * 75 = 1575$ долара.
- Годишен доход = $1575 * 12 + 1575 * 2.5 = 22837.5$ долара.
- Данък = 25% от 22837.5 = 5709.375 лева.
- Чист годишен доход = 17128.125 долара = 27233.71875 лева.
- Средна печалба на ден = $27233.71875 / 365 = 74.61$ лева.

Насоки и подсказки

Първо да анализираме задачата и да измислим как да я решим. След това ще изберем типовете данни и накрая ще напишем кода на решението.

Идея за решение

Нека първо пресметнем **колко е месечната заплата** на Иван. Това ще направим като **умножим работните дни в месеца по парите**, които той печели на ден. **Умножаваме получения резултат** първо по 12, за да изчислим колко е заплатата му за 12 месеца, а след това и **по 2.5**, за да пресметнем бонуса. Като съберем двете получени стойности, ще изчислим **общия му годишен доход**. От него **трябва да извадим 25%**. Това може да направим като умножим общия доход по **0.25** и извадим резултата от него. Спрямо дадения ни курс **преобразуваме долларите в лева**, след което **разделяме резултата на дните в годината**, за които приемаме, че са 365.

Избор на типове данни

Работните дни за месец са дадени като **цяло число**, следователно за тяхната стойност може да декларираме променлива от **тип int**. За **изкараните пари**, както и за курса на **долара спрямо лева**, ще получим реално **число**, следователно за тях използваме **double**. Тъй като **double** е типът данни с **по-голям обхват**, а за изходната стойност също се изисква **реално число** (с цяла и дробна част), ще използваме него и за останалите променливи, които създаваме.

Решение

Отново: след като имаме идея как да решим задачата и сме помислили за типовете данни, с които ще работим, пристъпваме към **писането на програмата**. Както и в предходните задачи, можем да разделим решението на три подзадачи:

- Прочитане на **входните данни**.
- Извършване на **изчисленията**.
- Извеждане на **изход** на конзолата.

Декларираме променливите, които ще използваме, като отново се стараем да изберем **подходящи имена**. С функцията **Console.ReadLine(...)** четем подадените числа на конзолата и **преобразуваме** въведения от потребителя стринг към **int** или **double** с **int,double.Parse(...)**.

Извършваме изчисленията:

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

Бихме могли да напишем израза, с който пресмятаме общия годишен доход, и без скоби. Тъй като умножението е операция с по-висок приоритет от събирането, то ще се извърши първо. Въпреки това **писането на скоби се препоръчва**, когато използваме повече оператори, защото така кодът става по-лесно четим и възможността да се допусне грешка е по-малка.

Накрая остава да изведем резултата на конзолата. Забелязваме, че се **изисква форматиране на числената стойност до втория знак след десетичната точка**. За целта можем да използваме placeholder, т.е. място, което ще бъде заместено с конкретна стойност при отпечатването. В C# за placeholder се използва цифра, оградена с къдрави скоби. Тъй като в програмирането броенето започва от 0, изразът **{0}** означава, че на негово място ще бъде поставен първият подаден аргумент. Цяло или дробно число можем да форматираме със спецификаторите F или f. След него следва цяло положително число, което указва броя на знаците след десетичния знак. Повече за форматирането може да прочетете от тук: http://introprogramming.info/intro-csharp-book/read-online/glava4-vhod-i-izhod-ot-konzolata/#_Toc298863992.

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

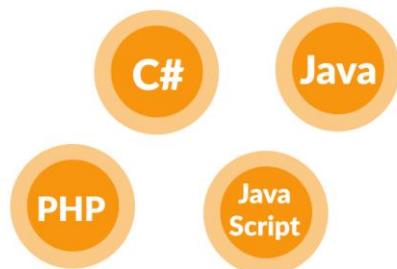
Тестване в Judge системата

Тествайте решението си: <https://judge.softuni.bg/Contests/Practice/Index/505#4>.

Качествено образование,
професия и работа за

Софтуерни инженери

- ✓ Безплатен старт за начинаещи - **присъствено и онлайн**
- ✓ Избор измежду **най-търсените професии** в софтуерната индустрия
- ✓ Съдействие за **кариерен старт**
- ✓ Отлични преподаватели, ментори и студентска общност



Софтуни предоставя съвременно иновативно образование за програмиране, ИТ и дигитални умения на хиляди млади хора. Програмата "**Софтуерен университет**" изгражда истински **професионалисти в света на програмирането**.

Включете се в цялостната програма по софтуерно инженерство на Софтуни, за да усвоите **най-търсените съвременни софтуерни технологии** чрез модерна учебна методика и с много практически задачи и проекти. **Учебният план** е разработен съвместно с ИТ фирмите и изгражда най-търсените от тях умения.

Софтуни работи пряко с компаниите от софтуерната индустрия, съдейтайки на своите студенти за реализацията им като успешни софтуерни инженери.

ПЪТЬТА НА СТУДЕНТА В СОФТУНИ



1 - 1.5 години



1.5 - 2 години



Programming Basics

Кариерен Старт

Дипломиране

70/100 credits

80/100/150 credits

Кандидатствай

softuni.bg/apply

Глава 3.1. Прости проверки

В настоящата глава ще разгледаме **условните конструкции в езика C#**, чрез които нашата програма може да има различно действие, в зависимост от дадено условие. Ще обясним синтаксиса на условните оператори за проверки (**if** и **if-else**) с подходящи примери и ще видим в какъв диапазон живее една променлива (нейният **обхват**). Накрая ще разгледаме техники за **дебъгване**, чрез които постъпково да проследяваме пътя, който извървява нашата програма по време на своето изпълнение.

Видео

Гледайте **видео-урок** по тази глава: https://youtube.com/watch?v=uwW_ueaOt7M.

Сравняване на числа

В програмирането можем да сравняваме стойности чрез следните **оператори**:

- Оператор `<` (по-малко)
- Оператор `>` (по-голямо)
- Оператор `<=` (по-малко или равно)
- Оператор `>=` (по-голямо или равно)
- Оператор `==` (равно)
- Оператор `!=` (различно)

При сравнение резултатът е булева стойност – **true** или **false**, в зависимост от това дали резултатът от сравнението е истина или лъжа.

Примери за сравнение на числа

```
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
```

Обърнете внимание, че при отпечатване на стойностите **true** и **false** в езика C#, те се отпечатват с главна буква, съответно **True** и **False**.

Оператори за сравнение

В езика C# можем да използваме следните оператори за сравнение на данни:

| Оператор | Означение | Работи за |
|-----------------------|--------------------|------------------------------------|
| Проверка за равенство | <code>==</code> | |
| Проверка за различие | <code>!=</code> | числа, стрингове, дати |
| По-голямо | <code>></code> | |
| По-голямо или равно | <code>>=</code> | числа, дати, други сравними типове |
| По-малко | <code><</code> | |
| По-малко или равно | <code><=</code> | |

Ето един пример:

```
var result = (5 <= 6);
Console.WriteLine(result); // True
```

Прости проверки

В програмирането често проверяваме **дадени условия** и извършваме различни действия, според резултата от проверката. Това става чрез проверката **if**, която има следната конструкция:

```
if (булев израз)
{
    // тяло на условната конструкция;
}
```

Пример: отлична оценка

Въвеждаме оценка в конзолата и проверяваме дали тя е отлична (≥ 5.50).

```
var grade = double.Parse(Console.ReadLine());
if (grade >= 5.50)
{
    Console.WriteLine("Excellent!");
}
```

Тествайте кода от примера локално с [Ctrl+F5]. Опитайте да въведете различни оценки, например 4.75, 5.49, 5.50 и 6.00. При оценки по-малки от 5.50 програмата няма да изведе нищо, а при оценка 5.50 или по-голяма, ще изведе “Excellent!”.

Тестване в Judge системата

Тествайте програмата от примера в judge системата на СофтУни:
<https://judge.softuni.bg/Contests/Practice/Index/506#0>.

Проверки с if-else конструкция

Конструкцията **if** може да съдържа и **else** клауза, с която да окажем конкретно действие в случай, че булевият израз (който е зададен в началото **if (булев израз)**) върне отрицателен резултат (**false**). Така построена, **условната конструкция** наричаме **if-else** и поведението ѝ е следното: ако резултатът от условието е **позитивен (true)** – извършваме едни действия, а когато е **негативен (false)** – други. Форматът на конструкцията е:

```
if (булево условие)
{
    // тяло на условната конструкция;
}
else
{
    // тяло на else-конструкция;
}
```

Пример: отлична оценка или не

Подобно на горния пример, въвеждаме оценка, проверяваме дали е отлична, но изписваме резултат и в двата случая.

```
var grade = double.Parse(Console.ReadLine());
if (grade >= 5.50)
{
    Console.WriteLine("Excellent!");
}
else
{
    Console.WriteLine("Not excellent.");
}
```

Тестване в Judge системата

Тествайте решението си: <https://judge.softuni.bg/Contests/Practice/Index/506#1>.

За къдравите скоби { } след if / else

Когато имаме само една команда в тялото на **if** конструкцията, можем да пропуснем къдравите скоби, обозначаващи тялото на условияния оператор. Когато искаем да изпълним блок от код (група команди), къдравите скоби са задължителни. В случай че ги изпуснем, ще се изпълни само първият ред след **if** клаузата.



Добра практика е, **винаги да слагаме къдрави скоби**, понеже това прави кода ни по-четим и по-подреден.

Ето един пример, в който изпускането на къдравите скоби води до объркване:

```
var color = "red";
if (color == "red")
    Console.WriteLine("tomato");
else if (color == "yellow")
    Console.WriteLine("banana");
Console.WriteLine("lemon");
```

Изпълнението на горния код ще изведе следния резултат на конзолата:

```
C:\WINDOWS\system3...
tomato
lemon
Press any key to continue . . .
```

С къдрави скоби:

```
var color = "red";
if (color == "red")
{
    Console.WriteLine("tomato");
}
else if (color == "yellow")
{
    Console.WriteLine("banana");
    Console.WriteLine("lemon");
}
```

На конзолата ще бъде отпечатано следното:

```
C:\WINDOWS\system...
tomato
banana
lemon
Press any key to continue . . .
```

Пример: четно или нечетно

Да се напише програма, която проверява, дали дадено цяло число е **четно** (even) или **нечетно** (odd).

Задачата можем да решим с помощта на една **if-else** конструкция и оператора **%**, който връща **остатък при деление** на две числа.

```
var num = int.Parse(Console.ReadLine());
if (num % 2 == 0)
{
    Console.WriteLine("even");
}
else
{
    Console.WriteLine("odd");
}
```

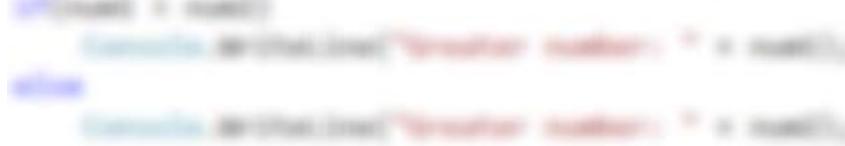
Тестване в Judge системата

Тествайте решението си: <https://judge.softuni.bg/Contests/Practice/Index/506#2>.

Пример: по-голямото число

Да се напише програма, която чете две цели числа и извежда по-голямото от тях. Първата ни задача е да прочетем двете числа. След което, чрез приструка **if-else** констукция, в съчетание с **оператора за по-голямо (>)**, да направим проверка. Част от кода е замъглено умислено, за да изprobваме наученото до момента.

```
Console.WriteLine("Enter two integers:");
var num1 = int.Parse(Console.ReadLine());
var num2 = int.Parse(Console.ReadLine());
```



Тестване в Judge системата

Тествайте решението си: <https://judge.softuni.bg/Contests/Practice/Index/506#3>.

Живот на променлива

Всяка една променлива си има обхват, в който съществува, наречен **variable scope**. Този обхват уточнява къде една променлива може да бъде използвана. В езика C# областта, в която една променлива съществува, започва от реда, на който сме я **дeфинириали** и завършва до първата затваряща къдрава скоба **}** (на метода, на **if** констукцията и т.н.). За това е важно да знаем, че **всяка променлива, дефинирана вътре в тялото на if, няма да бъде достъпна извън него**, освен ако не сме я дефинириали по-нагоре в кода.

В примера по-долу, на последния ред, на който се опитваме да отпечатаме променливата **salary**, която е дефинирана в **if** конструкцията, ще получим грешка, защото нямаме достъп до нея.

```
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!
```

Серии от проверки

Понякога се налага да извършим серия от проверки, преди да решим какви действия ще изпълнява нашата програма. В такива случаи, можем да приложим конструкцията **if-else if...-else** в **серия**. За целта използваме следния формат:

```
if (условие)
{
    // тяло на условната конструкция;
}
else if (условие2)
{
    // тяло на условната конструкция;
}
else if (условие3)
{
    // тяло на условната конструкция;
}
...
else
{
    // тяло на else-конструкция;
}
```

Пример: число от 1 до 9 на английски

Да се изпише число в интервала от 1 до 9 с текст на английски език (числото се чете от конзолата). Можем да прочетем числото и след това чрез **серия от проверки** отпечатваме съответстващата му английска дума:

```

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");
}

```

Програмната логика от примера по-горе последователно сравнява входното число от конзолата с цифрите от 1 до 9, като всяко следващо сравнение се извършва, само в случай че предходното сравнение не е било истина. В крайна сметка, ако никое от **if** условията не е изпълнено, се изпълнява последната **else** клауза.

Тестване в Judge системата

Тествайте решението си: <https://judge.softuni.bg/Contests/Practice/Index/506#4>.

Упражнения: прости проверки

За да затвърдим знанията си за условните конструкции **if** и **if-else**, ще решим няколко практически задачи.

Задача: бонус точки

Дадено е **цяло число** – брой точки. Върху него се начисляват **бонус точки** по правилата, описани по-долу. Да се напише програма, която пресмята **бонус точките** за това число и **общия брой точки** с бонусите.

- Ако числото е **до 100** включително, бонус точките са 5.
- Ако числото е **по-голямо от 100**, бонус точките са **20%** от числото.
- Ако числото е **по-голямо от 1000**, бонус точките са **10%** от числото.

- Допълнителни бонус точки (начисляват се отделно от предходните):
 - За четно число -> + 1 т.
 - За число, което завършва на 5 -> + 2 т.

Примерен вход и изход

| Вход | Изход | Вход | Изход | Вход | Изход | Вход | Изход |
|------|---------|------|-----------|------|-----------------|-------|-------------------|
| 20 | 6 26 | 175 | 37 212 | 2703 | 270.3 2973.3 | 15875 | 1589.5 17464.5 |

Насоки и подсказки

Основните и допълнителните бонус точки можем да изчислим с поредица от няколко **if-else-if-else** проверки. Като за основните бонус точки имаме 3 случая (когато въведеното число е до 100, между 100 и 1000 и по-голямо от 1000), а за допълнителните бонус точки – още 2 случая (когато числото е четно и нечетно).

```
var num = int.Parse(Console.ReadLine());
var bonusScore = 0.0;

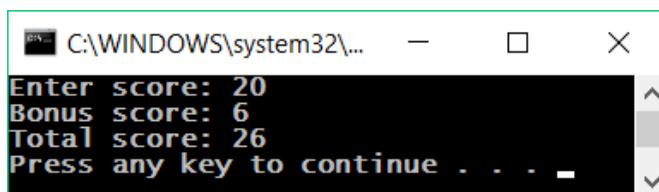
if (num > 1000)
{
    bonusScore = num * 0.10;
}
else // TODO: Write more logic here...

if (num % 10 == 5)
{
    bonusScore += 2;
}
else // TODO: Write more logic here...

// bonus score:
Console.WriteLine(bonusScore);

// total score:
Console.WriteLine(num + bonusScore);
```

Ето как би могло да изглежда решението на задачата в действие:



Обърнете внимание, че за тази задача judge е настроен да игнорира всичко, което не е число, така че можем да печатаме не само числата, но и уточняващ текст.

Тестване в Judge системата

Тествайте решението си: <https://judge.softuni.bg/Contests/Practice/Index/506#5>.

Задача: сумиране на секунди

Трима спортни състезатели финишират за някакъв **брой секунди** (между 1 и 50). Да се напише програма, която въвежда времената на състезателите и пресмята **сумарното им време** във формат "минути:секунди". Секундите да се изведат с **водеща нула** (2 -> "02", 7 -> "07", 35 -> "35").

Примерен вход и изход

| Вход | Изход |
|------|-------|
| 35 | |
| 45 | 2:04 |
| 44 | |
| 50 | |
| 50 | 2:29 |
| 49 | |

| Вход | Изход |
|------|-------|
| 22 | |
| 7 | 1:03 |
| 34 | |
| 14 | |
| 12 | 0:36 |
| 10 | |

Насоки и подсказки

Първо сумираме трите числа, за да получим общия резултат в секунди. Понеже **1 минута = 60** секунди, ще трябва да изчислим броя минути и броя секунди в диапазона от 0 до 59:

- Ако резултатът е между 0 и 59, отпечатваме 0 минути + изчислените секунди.
- Ако резултатът е между 60 и 119, отпечатваме 1 минута + изчислените секунди минус 60.
- Ако резултатът е между 120 и 179, отпечатваме 2 минути + изчислените секунди минус 120.
- Ако секундите са по-малко от 10, изваждаме водеща нула преди тях.

Следва примерна имплементация на описаната идея:

```
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);
}

```

Тестване в Judge системата

Тествайте решението си: <https://judge.softuni.bg/Contests/Practice/Index/506#6>.

Задача: конвертор за мерни единици

Да се напише програма, която преобразува разстояние между следните **8 мерни единици**: **m, mm, cm, mi, in, km, ft, yd**. Използвайте съответствията от таблицата по-долу:

| Входна единица | Изходна единица |
|----------------|---------------------------|
| 1 meter (m) | 1000 millimeters (mm) |
| 1 meter (m) | 100 centimeters (cm) |
| 1 meter (m) | 0.000621371192 miles (mi) |
| 1 meter (m) | 39.3700787 inches (in) |
| 1 meter (m) | 0.001 kilometers (km) |
| 1 meter (m) | 3.2808399 feet (ft) |
| 1 meter (m) | 1.0936133 yards (yd) |

Входните данни се състоят от три реда:

- Първи ред: число за преобразуване.
- Втори ред: входна мерна единица.
- Трети ред: изходна мерна единица (за резултата).

Примерен вход и изход

| Вход | Изход | Вход | Изход | Вход | Изход |
|----------------|------------|-----------------|------------------|-----------------|------------------|
| 12 km ft | 39370.0788 | 150 mi in | 9503999.99393599 | 450 yd km | 0.41147999937455 |

Насоки и подсказки

Прочитаме си входните данни, като към прочитането на мерните единици можем да добавим функцията **ToLower()**, която ще направи буквите малки. Както виждаме от таблицата в условието, можем да конвертираме само **между метри и някаква друга мерна единица**. Следователно трябва първо да изчислим числото за преобразуване в метри. Затова трябва да направим набор от проверки, за да определим каква е входната мерна единица, а след това и за изходната мерна единица.

```
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);
```

Тестване в Judge системата

Тествайте решението си: <https://judge.softuni.bg/Contests/Practice/Index/506#7>.

Дебъгване – прости операции с дебъгер

До момента писахме доста код и често пъти в него имаше грешки, нали? Сега ще покажем един инструмент, с който можем да намираме грешките по-лесно.

Какво е "дебъгване"?

Дебъгване е процесът на „закачане“ към изпълнението на програмата, който ни позволява да проследяваме нейното изпълнение. Извършва се с **дебъгер**:

```

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 }
13

```

The screenshot shows the Visual Studio IDE. In the code editor, a green vertical bar indicates the current line of execution (line 9). A blue box highlights the line of code: `number += number;`. Below the code editor, the status bar shows "84 %". A blue-bordered window titled "Locals" is open, displaying the following table:

| Name | Value |
|----------------------------------|-------|
| System.Console.ReadLine returned | "10" |
| int.Parse returned | 10 |
| number | 10 |

Можем да следим **ред по ред** какво се случва с нашата програма, какъв път следва, какви стойности имат дефинираните променливи на всяка стъпка от дебъгването и много други неща, които ни позволяват да откриваме грешки (бъгове).

Дебъгване във Visual Studio

Чрез натискане на бутона **[F10]**, стартираме програмата в **debug режим**. Преминаваме към **следващия ред** отново с **[F10]**.

Чрез **[F9]** създаваме стопери – така наречените **breakpoints**, до които можем да стигнем директно използвайки **[F5]** при стартирането на програмата.

```

1     using System;
2
3     namespace Simple_Conditions
4     {
5         class Program
6         {
7             static void Main(string[] args)
8             {
9
10                int five = 5;
11                if (five == 5)
12                {
13                    Console.WriteLine("Yes");
14                }
15            }
16        }
17    }

```

Call Stack Breakpoints Exception Settings Command Window Immediate Window Output Autos Locals Watch 1

Ready Ln 11 Col 27 Ch 27 INS Add to Source Control

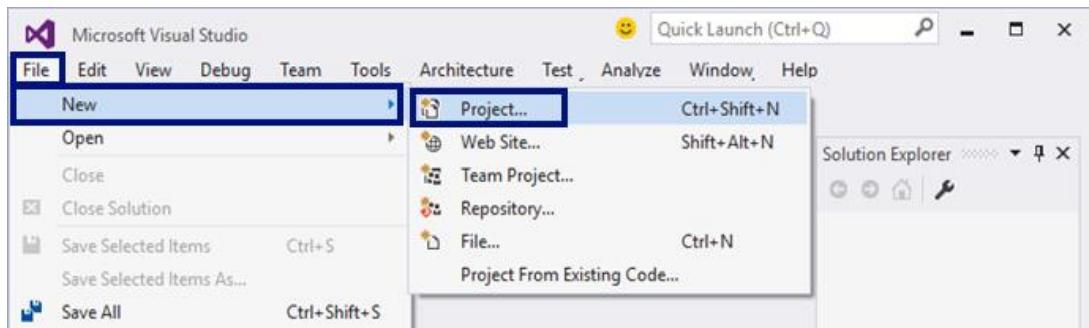
Упражнения: прости проверки

Нека затвърдим наученото в тази глава с няколко задачи.

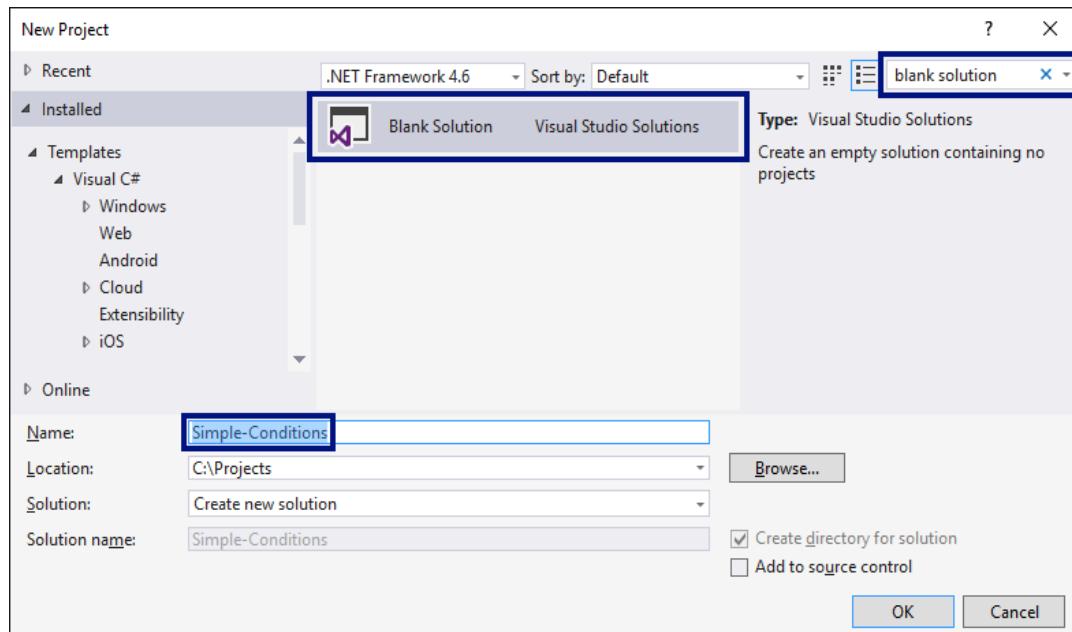
Празно Visual Studio решение (Blank Solution)

Първо създаваме празно решение (Blank Solution) във Visual Studio, за да организираме по-добре решенията на задачите от упражненията – всяка задача ще бъде в отделен проект и всички проекти ще бъдат в общ solution.

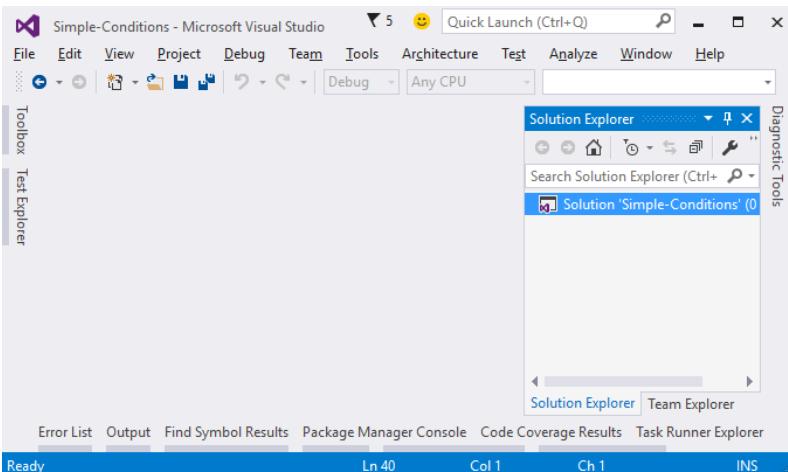
Стартираме Visual Studio. Създаваме нов Blank Solution: [File] -> [New] -> [Project].



Избираме от диалоговия прозорец [Templates] -> [Other Project Types] -> [Visual Studio Solutions] -> [Blank Solution] и даваме подходящо име на проекта, например "Simple-Conditions":



Сега имаме създаден празен Visual Studio Solution (без проекти в него):



Задача: проверка за отлична оценка

Първата задача от упражненията за тази тема е да се напише конзолна програма, която въвежда оценка (десетично число) и отпечатва "Excellent!", ако оценката е 5.50 или по-висока.

Примерен вход и изход

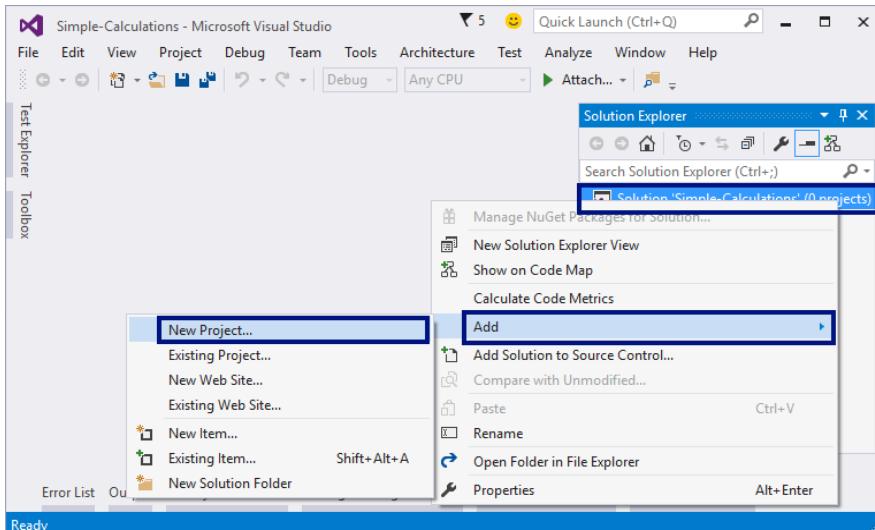
| Вход | Изход |
|------|------------|
| 6 | Excellent! |

| Вход | Изход |
|------|------------|
| 5.5 | Excellent! |

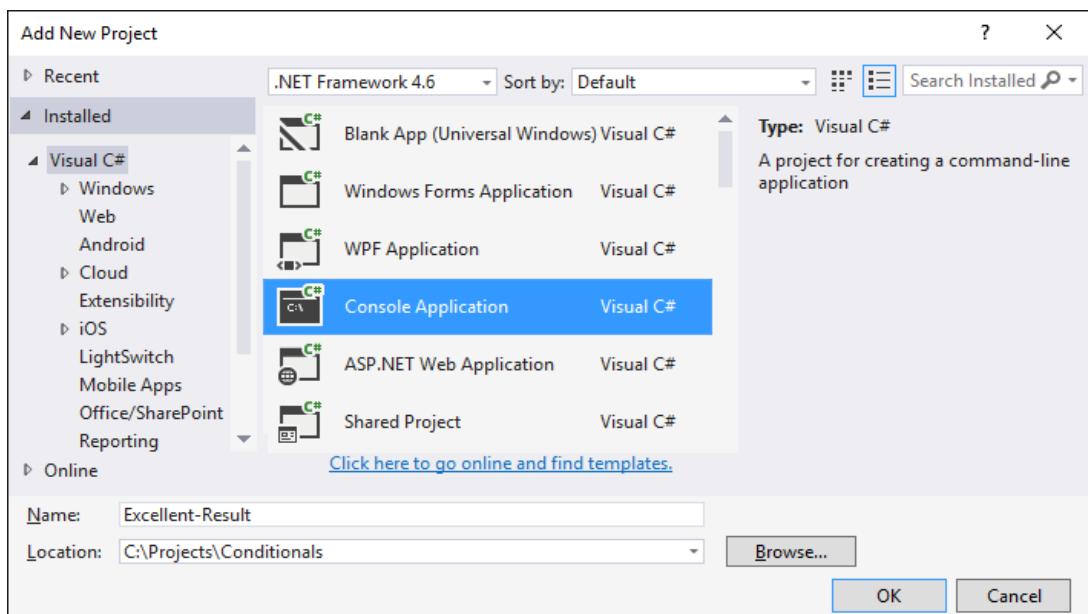
| Вход | Изход |
|------|--------------|
| 5.49 | (няма изход) |

Насоки и подсказки

Създаваме **нов проект** в съществуващото Visual Studio решение. В Solution Explorer кликваме с десен бутон на мишката върху **Solution 'Simple-Conditions'**. Избираме **[Add] -> [New Project]**:



Ще се отвори диалогов прозорец за избор на тип проект за създаване. Избираме **C# конзолно приложение** и задаваме име, например “**Excellent-Result**”:



Вече имаме solution с едно конзолно приложение в него. Остава да напишем кода за решаване на задачата.

За целта отиваме в тялото на метода **Main(string[] args)** и пишем следния код:

```
namespace Excellent_Result
{
    class Program
    {
        static void Main(string[] args)
        {
            var grade = double.Parse(Console.ReadLine());
            if (grade >= 5.50)
            {
                Console.WriteLine("Excellent!");
            }
        }
    }
}
```

Стартираме програмата с [Ctrl+F5], за да я **тестваме** с различни входни стойности:

C:\WINDOWS\system32\cmd.exe

5.25
Press any key to continue . . .

C:\WINDOWS\system32\cmd.e...

5.6
Excellent!
Press any key to continue . . .

Тестване в Judge системата

Тествайте решението си в Judge системата:

<https://judge.softuni.bg/Contests/Practice/Index/506#0>.

Conditional Statements - <https://judge.softuni.bg/Contests/Practice/Index/152#0>

Excellent Result

Participants Tests Change Delete

Administration |

```

4 using System.Text;
5 using System.Threading.Tasks;
6
7 namespace Excellent_Result
8 {
9     class Program
10    {
11        static void Main(string[] args)
12        {
13            var grade = double.Parse(Console.ReadLine());
14            if (grade >= 5.50)
15            {
16                Console.WriteLine("Excellent!");
17            }
18        }
19    }
20 }
21

```

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

Allowed memory: 16.00 MB

| Submissions | | |
|-------------------|----------------------------------|---|
| Points | Time and memory used | Submission date |
| ✓✓✓✓✓✓✓ 100 / 100 | Memory: 7.81 MB Time: 0.029 s | 17:51:38 26.01.2016 Details |

Задача: отлична оценка или не

Следващата задача от тази тема е да се напише конзолна програма, която въвежда оценка (десетично число) и отпечатва "Excellent!", ако оценката е 5.50 или по-висока, или "Not excellent." в противен случай.

Примерен вход и изход

| Вход | Изход |
|------|------------|
| 6 | Excellent! |

| Вход | Изход |
|------|----------------|
| 5 | Not Excellent! |

| Вход | Изход |
|------|----------------|
| 5.49 | Not excellent. |

Насоки и подсказки

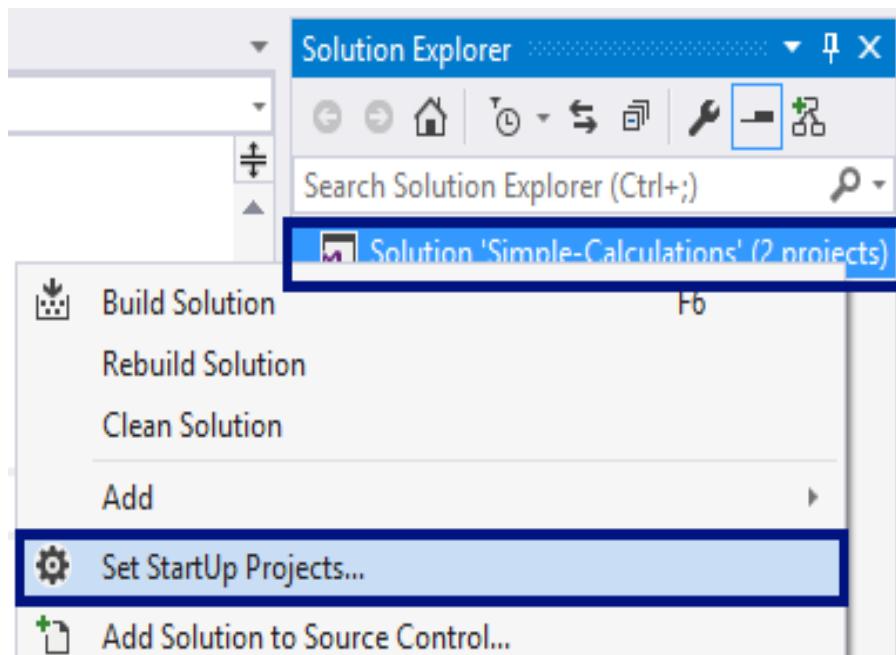
Първо създаваме нов C# конзолен проект в решението "Simple-Conditions".

- Кликаме с мишката върху решението в Solution Explorer и избираме [Add] -> [New Project].
- Избираме [Visual C#] -> [Windows] -> [Console Application] и задаваме име "Excellent-or-Not".

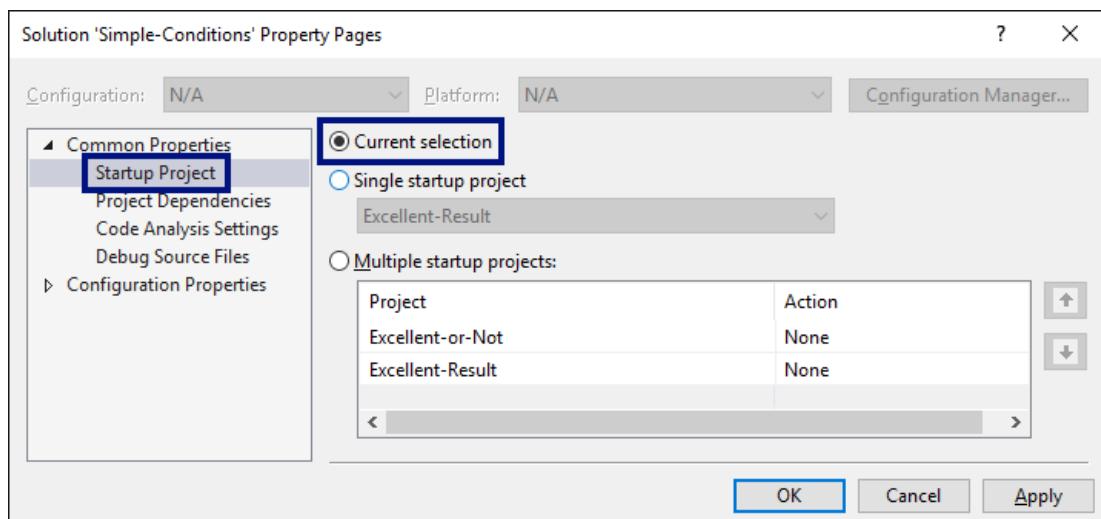
Следва да **напишем кода** на програмата. Може да си помогнем с примерния код от картинката:

```
var grade = double.Parse(Console.ReadLine());
if (grade >= 5.50)
{
    Console.WriteLine("Excellent!");
}
else
{
    Console.WriteLine("Not excellent.");
}
```

Включваме режим на **автоматично превключване** към текущия проект като кликнем върху главния solution с десния бутон на мишката и изберем [Set StartUp Projects...]:



Ще се появи диалогов прозорец, от който трябва да се избере [Startup Project] -> [Current selection]:



Сега **стартираме програмата**, както обикновено с [Ctrl+F5] и я тестваме дали работи коректно:

```
C:\WINDOWS\system32\cmd.e... 5.6 Excellent!
C:\WINDOWS\system32\cmd.exe 4.25 Not excellent.
```

Тестване в Judge системата

Тествайте решението си: <https://judge.softuni.bg/Contests/Practice/Index/506#1>.

| Submissions | | |
|-------------|----------------------------------|---------------------|
| 1 | | |
| Points | Time and memory used | Submission date |
| 100 / 100 | Memory: 7.83 MB Time: 0.014 s | 22:37:59 21.01.2016 |

Задача: четно или нечетно

Да се напише програма, която въвежда **цяло число** и печата дали е **четно** или **нечетно**.

Примерен вход и изход

| Вход | Изход |
|------|-------|
| 2 | even |

| Вход | Изход |
|------|-------|
| 3 | odd |

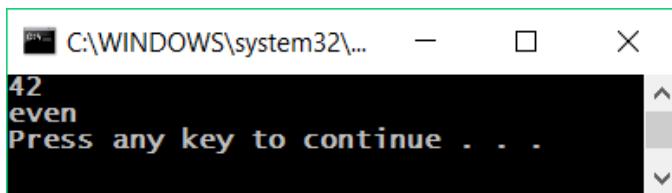
| Вход | Изход |
|------|-------|
| 25 | odd |

| Вход | Изход |
|------|-------|
| 25 | odd |

Насоки и подсказки

Отново, първо добавяме **нов C# конзолен проект** в съществуващия solution. В метода **static void Main()** трябва да напишем, кода на програмата. Проверката дали дадено число е четно, може да се реализира с оператора **%**, който ще ни върне остатъка при целочислено деление на 2 по следния начин: **var isEven = (num % 2 == 0)**.

Остава да **стартираме** програмата с [Ctrl+F5] и да я тестваме:



Тестване в Judge системата

Тествайте решението си: <https://judge.softuni.bg/Contests/Practice/Index/506#2>.

Задача: намиране на по-голямото число

Да се напише програма, която въвежда **две цели числа** и отпечатва по-голямото от двете.

Примерен вход и изход

| Вход | Изход |
|--------|-------|
| 5 3 | 5 |

| Вход | Изход |
|--------|-------|
| 3 5 | 5 |

| Вход | Изход |
|----------|-------|
| 10 10 | 10 |

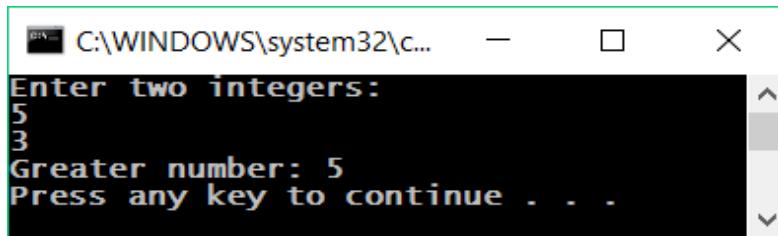
| Вход | Изход |
|---------|-------|
| -5 5 | 5 |

Насоки и подсказки

Както обикновено, първо трябва да добавим **нов C# конзолен проект** в съществуващия solution. За кода на програмата ни е необходима единична **if-else** конструкция. Може да си помогнете частично с кода от картинката, който е умишлено замъглен, за да помислите как да го допишете сами:

```
Console.WriteLine("Enter two integers:");
var num1 = int.Parse(Console.ReadLine());
var num2 = int.Parse(Console.ReadLine());
```

След като сме готови с имплементацията на решението, **стартираме** програмата с [Ctrl+F5] и я тестваме:



Тестване в Judge системата

Тествайте решението си: <https://judge.softuni.bg/Contests/Practice/Index/506#3>.

Задача: изписване на число до 9 с думи

Да се напише програма, която въвежда **цяло число в диапазона [0...9]** и го **изписва с думи** на английски език. Ако числото е извън диапазона, изписва съобщението "number too big".

Примерен вход и изход

| Вход | Изход | Вход | Изход | Вход | Изход | Вход | Изход |
|------|-------|------|-------|------|-------|------|----------------|
| 5 | five | 1 | one | 9 | nine | 10 | number too big |

Насоки и подсказки

Може да използваме поредица **if-else** конструкции, с които да разгледаме възможните **11 случая**.

Тестване в Judge системата

Тествайте решението си: <https://judge.softuni.bg/Contests/Practice/Index/506#4>.

Задача: познай паролата

Да се напише програма, която **въвежда парола** (един ред с произволен текст) и проверява дали въведеното **съвпада** с фразата “`s3cr3t!P@ssw0rd`”. При съответствие да се изведе “`Welcome`”, а при несъответствие да се изведе “`Wrong password!`”.

Примерен вход и изход

| Вход | Изход | Вход | Изход | Вход | Изход |
|--------|-----------------|-----------------|---------|-------------|-----------------|
| qwerty | Wrong password! | s3cr3t!P@ssw0rd | Welcome | s3cr3t!p@ss | Wrong password! |

Насоки и подсказки

Използвайте **if-else** конструкция.

Тестване в Judge системата

Тествайте решението си: <https://judge.softuni.bg/Contests/Practice/Index/506#8>.

Задача: число от 100 до 200

Да се напише програма, която **въвежда цяло число** и проверява дали е **под 100**, **между 100 и 200** или **над 200**. Да се отпечатат съответно съобщения, като в примерите по-долу.

Примерен вход и изход

| Вход | Изход | Вход | Изход | Вход | Изход |
|------|---------------|------|---------------------|------|------------------|
| 95 | Less than 100 | 120 | Between 100 and 200 | 210 | Greater than 200 |

Тестване в Judge системата

Тествайте решението си: <https://judge.softuni.bg/Contests/Practice/Index/506#9>.

Задача: еднакви думи

Да се напише програма, която **въвежда две думи** и проверява дали са еднакви. Да не се прави разлика между главни и малки букви. Да се изведе “`yes`” или “`no`”.

Примерен вход и изход

| Вход | Изход | Вход | Изход | Вход | Изход | Вход | Изход |
|----------------|-------|--------------------|-------|-------------|-------|---------------|-------|
| Hello Hello | yes | SoftUni softuni | yes | Soft Uni | no | beer vodka | no |

Насоки и подсказки

Преди сравняване на думите, ги обърнете в долен регистър, за да не оказва влияние размера на буквите (главни/малки): **word = word.ToLower()**.

Тестване в Judge системата

Тествайте решението си: <https://judge.softuni.bg/Contests/Practice/Index/506#10>.

Задача: информация за скоростта

Да се напише програма, която **въвежда скорост** (десетично число) и отпечатва **информация за скоростта**. При скорост до 10 (включително), отпечатайте "slow". При скорост над 10 и до 50, отпечатайте "average". При скорост над 50 и до 150, отпечатайте "fast". При скорост над 150 и до 1000, отпечатайте "ultra fast". При по-висока скорост, отпечатайте "extremely fast".

Примерен вход и изход

| Вход | Изход | Вход | Изход | Вход | Изход |
|------|---------|------|------------|------|----------------|
| 8 | slow | 126 | fast | 3500 | extremely fast |
| 49.5 | average | 160 | ultra fast | | |

Тестване в Judge системата

Тествайте решението си: <https://judge.softuni.bg/Contests/Practice/Index/506#11>.

Задача: лица на фигури

Да се напише програма, която **въвежда размерите на геометрична фигура и пресмята лицето ѝ**. Фигурите са четири вида: квадрат (**square**), правоъгълник (**rectangle**), кръг (**circle**) и триъгълник (**triangle**).

На първия ред на входа се чете вида на фигурата (**square**, **rectangle**, **circle**, **triangle**).

- Ако фигурата е **квадрат**, на следващия ред се чете едно число – дължина на страната му.
- Ако фигурата е **правоъгълник**, на следващите два реда се четат две числа – дълчините на страните му.
- Ако фигурата е **кръг**, на следващия ред се чете едно число – радиусът на кръга.
- Ако фигурата е **триъгълник**, на следващите два реда се четат две числа – дължината на страната му и дължината на височината към нея.

Резултатът да се закръгли до **3 цифри след десетичния знак**.

Примерен вход и изход

| Вход | Изход | Вход | Изход | Вход | Изход | Вход | Изход |
|-------------|-------|-----------------------|-------|-------------|---------|-----------------------|-------|
| square 5 | 25 | rectangle 7 2.5 | 17.5 | circle 6 | 113.097 | triangle 4.5 20 | 45 |

Тестване в Judge системата

Тествайте решението си: <https://judge.softuni.bg/Contests/Practice/Index/506#12>.

Задача: време + 15 минути

Да се напише програма, която **въвежда час и минути** от 24-часово денонощие и изчислява колко ще е **частът след 15 минути**. Резултатът да се отпечата във формат **hh:mm**. Часовете винаги са между 0 и 23, а минутите винаги са между 0 и 59. Часовете се изписват с една или две цифри. Минутите се изписват винаги с по две цифри (с **водеща нула** при едноцифрен число).

Примерен вход и изход

| Вход | Изход | Вход | Изход | Вход | Изход | Вход | Изход |
|---------|-------|---------|-------|----------|-------|----------|-------|
| 1 46 | 2:01 | 0 01 | 0:16 | 23 59 | 0:14 | 11 08 | 11:23 |

Насоки и подсказки

Добавете 15 минути и направете няколко проверки. Ако минутите надвишат 59, **увеличете часовете с 1** и **намалете минутите** с 60. По аналогичен начин разгледайте случая, когато часовете надвишат 23. При печатането на минутите, **проверете за водеща нула**.

Тестване в Judge системата

Тествайте решението си: <https://judge.softuni.bg/Contests/Practice/Index/506#13>.

Задача: еднакви 3 числа

Да се напише програма, в която се въвеждат **3 числа** и се отпечатва дали те са **еднакви** (**yes / no**).

Примерен вход и изход

| Вход | Изход | Вход | Изход | Вход | Изход |
|-------------|-------|-------------|-------|-------------|-------|
| 5 5 5 | yes | 5 4 5 | no | 1 2 3 | no |

Тестване в Judge системата

Тествайте решението си: <https://judge.softuni.bg/Contests/Practice/Index/506#14>.

Задача: * изписване на число от 0 до 100 с думи

Да се напише програма, която превръща число в диапазона [0...100] в текст.

Примерен вход и изход

| Вход | Изход |
|------|-------------|
| 25 | twenty five |

| Вход | Изход |
|------|-----------|
| 42 | forty two |

| Вход | Изход |
|------|-------|
| 6 | six |

Насоки и подсказки

Проверете първо за **едноцифриeni числа** и ако числото е едноцифрено, отпечатайте съответната дума за него. След това проверете за **двуцифриeni числа**. Тях отпечатавайте на две части: лява част (**десетици** = числото / 10) и дясна част (**единици** = числото % 10). Ако числото има 3 цифри, трябва да е 100 и може да се разгледа като специален случай.

Тестване в Judge системата

Тествайте решението си: <https://judge.softuni.bg/Contests/Practice/Index/506#15>.

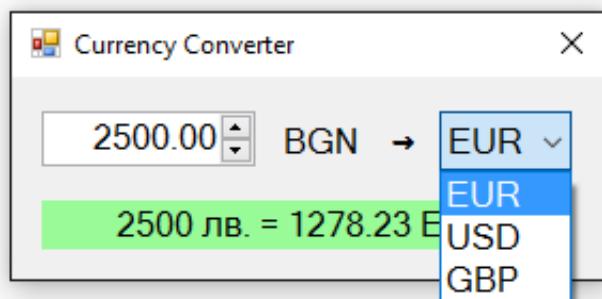
Графично (desktop) приложение

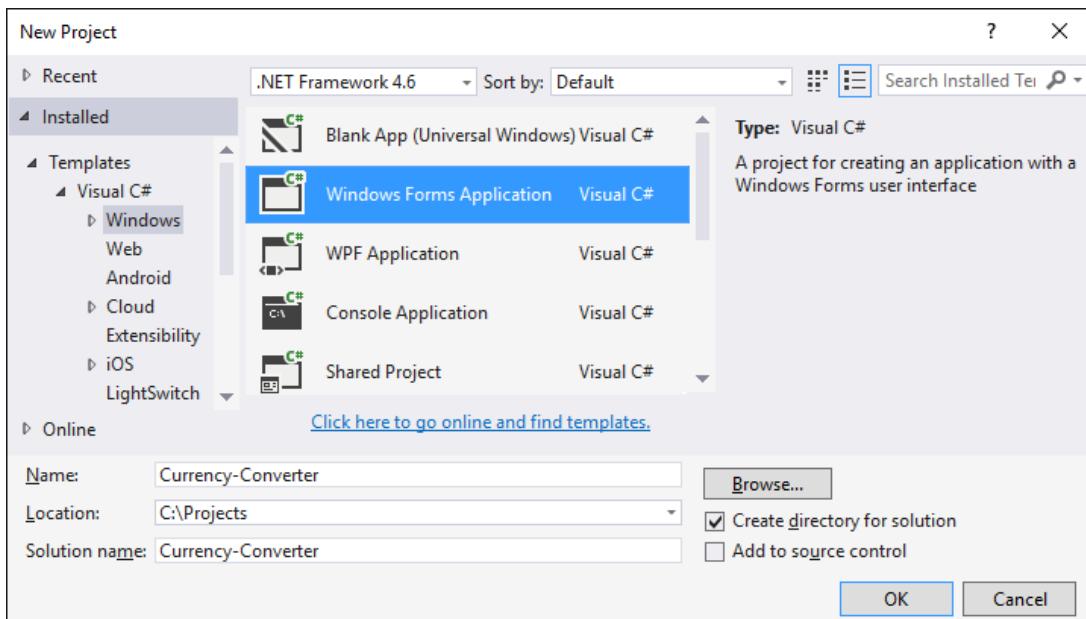
След като направихме няколко упражнения върху **условни конструкции (проверки)**, сега нека направим нещо по-интересно: приложение с графичен потребителски интерфейс за конвертиране на валути. Ще използваме знанията от тази глава, за да избираме измежду няколко налични валути и съответно да извършваме пресмятания по различен курс спрямо избраната валута.

Задача: конвертор за валути

Нека разгледаме как да създадем графично (GUI) приложение за **конвертиране на валути**. Приложението ще изглежда приблизително като на картинката вдясно.

Този път създаваме нов Windows Forms Application с име "Currency-Converter":

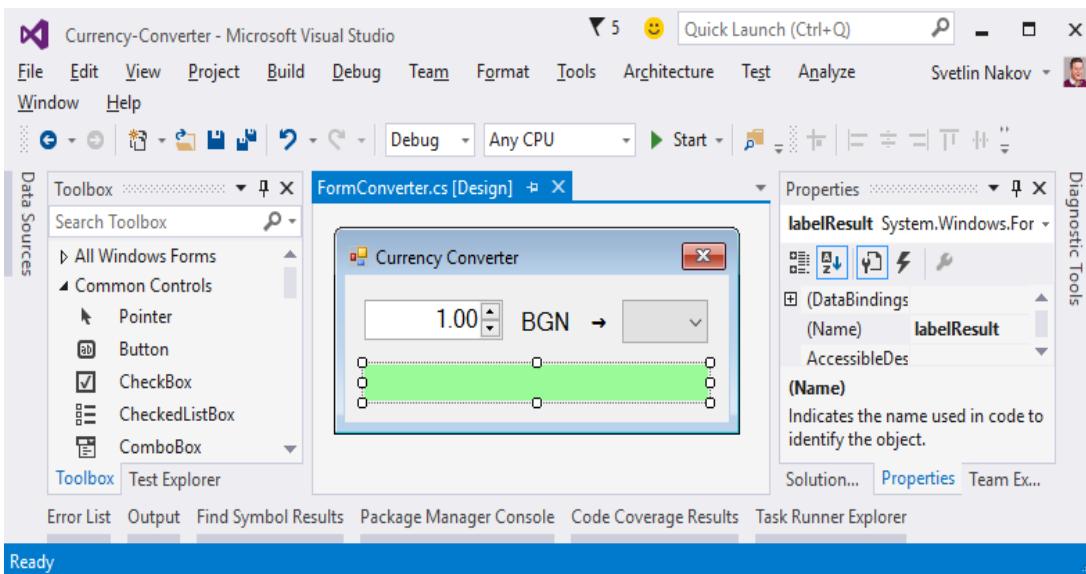




Нареждаме следните контроли във формата:

- Една кутийка за въвеждане на число (**NumericUpDown**)
- Един падащ списък с валути (**ComboBox**)
- Текстов блок за резултата (**Label**)
- Няколко надписа (**Label**)

Нагласяме **размерите** и свойствата им, за да изглеждат долу-горе като на картина-ката по-долу:

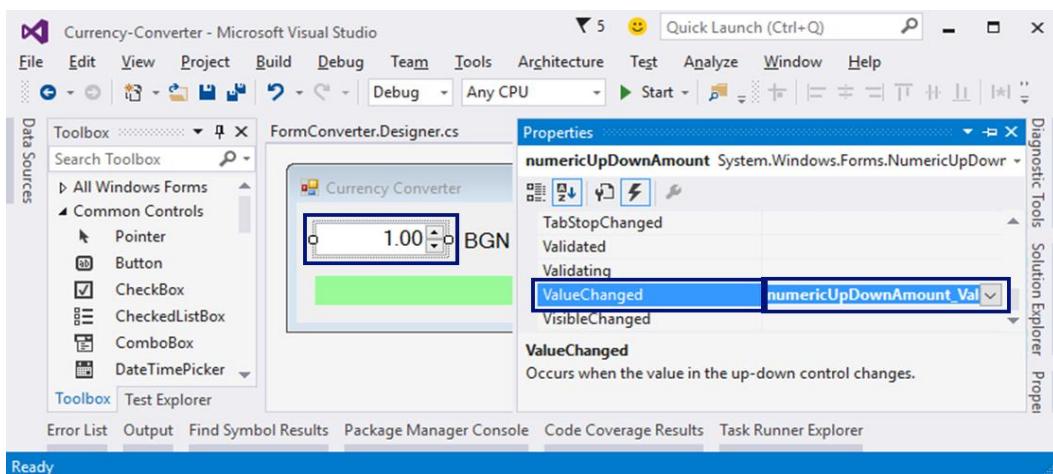


Задаваме следните **настройки** на контролите:

- За главната форма (**Form**), която съдържа всички контроли:
 - **(name)** = **FormConverter**
 - **Text** = "Currency Converter"
 - **Font.Size** = 12
 - **MaximizeBox** = False
 - **MinimizeBox** = False
 - **FormBorderStyle** = FixedSingle
- За полето за въвеждане на число (**NumericUpDown**):
 - **(name)** = **numericUpDownAmount**
 - **Value** = 1
 - **Minimum** = 0
 - **Maximum** = 1000000
 - **TextAlign** = Right
 - **DecimalPlaces** = 2
- За падащия списък с валутите (**ComboBox**):
 - **(name)** = **comboBoxCurrency**
 - **DropDownStyle** = DropDownList
 - **Items** =
 - EUR
 - USD
 - GBP
- За текстовия блок за резултата (**Label**):
 - **(name)** = **labelResult**
 - **AutoSize** = False
 - **BackColor** = PaleGreen
 - **TextAlign** = MiddleCenter
 - **Font.Size** = 14
 - **Font.Bold** = True

Трябва да хванем следните **събития**, за да напишем C# кода, който ще се изпълни при настъпването им:

- Събитието **ValueChanged** на контролата за въвеждане на число **numericUpDownAmount**:



- Събитието **Load** на формата **FormConverter**
- Събитието **SelectedIndexChanged** на падащия списък за избор на валута **comboBoxCurrency**

Ще използваме следния C# код за обработка на събитията:

```
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 sender, EventArgs e)
{
    ConvertCurrency();
}
```

Задачата на горния код е да избере при стартиране на програмата валута “EUR” и при промяна на стойностите в полето за сума или при смяна на валутата, да изчисли резултата, извиквайки **ConvertCurrency()** метода.

Следва да напишем действието **ConvertCurrency()** за конвертиране на въведена сумата от лева в избраната валута:

```
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 + " лв. = " +
Math.Round(convertedAmount, 2) + " " +
this.comboBoxCurrency.SelectedItem;
}
```

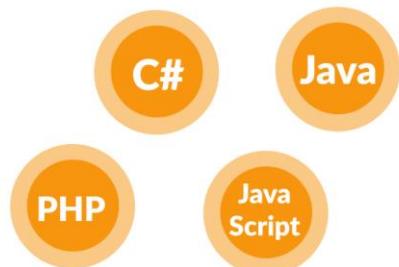
Горният код взима **сумата** за конвертиране от полето **numericUpDownAmount** и избраната валута за резултата от полето **comboBoxCurrency**. След това с **условна конструкция**, според избраната валута, сумата се дели на **валутния курс** (който е фиксиран твърдо в сурс кода). Накрая се генерира текстово **съобщение с резултата** (закръглен до 2 цифри след десетичния знак) и се записва в зелената кутийка **labelResult**. Опитайте!

Ако имате проблеми с примера по-горе, **гледайте видеото** в началото на тази глава или питайте във **форума на СофтУни**: <https://softuni.bg/forum>.

Качествено образование,
професия и работа за

Софтуерни инженери

- ✓ Безплатен старт за начинаещи - **присъствено и онлайн**
- ✓ Избор измежду **най-търсените професии** в софтуерната индустрия
- ✓ Съдействие за **кариерен старт**
- ✓ Отлични преподаватели, ментори и студентска общност



Софтуни предоставя съвременно иновативно образование за програмиране, ИТ и дигитални умения на хиляди млади хора. Програмата "**Софтуерен университет**" изгражда истински **професионалисти в света на програмирането**.

Включете се в цялостната програма по софтуерно инженерство на Софтуни, за да усвоите **най-търсените съвременни софтуерни технологии** чрез модерна учебна методика и с много практически задачи и проекти. **Учебният план** е разработен съвместно с ИТ фирмите и изгражда най-търсените от тях умения.

Софтуни работи пряко с компаниите от софтуерната индустрия, съдейтайки на своите студенти за реализацията им като успешни софтуерни инженери.

ПЪТЯТ НА СТУДЕНТА В СОФТУНИ



1 - 1.5 години



1.5 - 2 години



Programming Basics

Кариерен Старт

Дипломиране

70/100 credits

80/100/150 credits

Кандидатствай

softuni.bg/apply

Глава 3.2. Прости проверки – изпитни задачи

В предходната глава разглеждахме **условните конструкции** в езика C#, чрез които можем да изпълняваме различни действия в зависимост от някакво условие. Споменахме още какъв е **обхватът** на една променлива (нейният **scope**), както и как постъпково да проследяваме изпълнението на нашата програма (т.нар. **дебъгване**). В настоящата глава ще упражним работата с логически проверки, като разглеждаме някои **задачи, давани по приемни изпити в СофтУни**. За целта нека първо си припомним конструкцията на логическата проверка:

```
if (булев израз)
{
    // тяло на условната конструкция;
}
else
{
    // тяло на else-конструкция;
}
```

if проверките се състоят от:

- **if** клауза
- булев израз – променлива от булев тип (**bool**) или булев логически израз (израз, който връща резултат **true/false**)
- тяло на конструкцията – съдържа произволен блок със сорс код
- **else** клауза и нейният блок със сорс код (**незадължително**)

Изпитни задачи

След като си припомнихме как се пишат условни конструкции, да решим няколко задачи, за да получим практически опит с **if-else**-конструкцията.

Задача: цена за транспорт

Студент трябва да пропътува **n** километра. Той има избор между **три вида транспорт**:

- **Такси.** Начална такса: **0.70** лв. Дневна тарифа: **0.79** лв./км. Нощна тарифа: **0.90** лв./км.
- **Автобус.** Дневна / нощна тарифа: **0.09** лв./км. Може да се използва за разстояния минимум **20** км.
- **Влак.** Дневна / нощна тарифа: **0.06** лв./км. Може да се използва за разстояния минимум **100** км.

Напишете програма, която въвежда броя **километри n** и **период от деня** (ден или нощ) и изчислява **цената на най-евтиния транспорт**.

Входни данни

От конзолата се четат **два реда**:

- Първият ред съдържа числото **n** – брой километри – цяло число в интервала [1...5000].
- Вторият ред съдържа дума “**day**” или “**night**” – пътуване през деня или през нощта.

Изходни данни

Да се отпечата на конзолата **най-ниската цена** за посочения брой километри.

Примерен вход и изход

| Вход | Изход | Вход | Изход | Вход | Изход | Вход | Изход |
|----------|-------|------------|-------|-----------|-------|--------------|-------|
| 5 day | 4.65 | 7 night | 7 | 25 day | 2.25 | 180 night | 10.8 |

Насоки и подсказки

Ще прочетем входните данни и в зависимост от разстоянието ще изберем най-евтиния транспорт. За целта ще използваме няколко проверки.

Обработка на входните данни

В условието на задачата е дадена **информация за входа и изхода**. Съответно, първите **два реда** от решението ще съдържат декларирането и инициализирането на двете **променливи**, в които ще пазим **стойностите на входните данни**.

- За **първия ред е упоменато**, че съдържа **цяло число**, затова и променливата, която ще бъде декларирана, е от тип **int**.
- За **втория ред** указането е, че съдържа **дума**, съответно променливата е от тип **string**.

```
int distance = int.Parse(Console.ReadLine());
string dayOrNight = Console.ReadLine();
```

Преди да започнем проверките е нужно да **декларираме** и една **променлива**, в която ще пазим стойността на **цената за транспорт**.

```
double price = 0;
```

Извършване на проверки и съответните изчисления

След като вече сме **декларирали и инициализирали** входните данни и променливата, в която ще пазим стойността на цената, трябва да преценим кои **условия** от задачата първо ще бъдат проверени.

От условието е видно, че тарифите на две от превозните средства **не зависят** от това дали е **ден** или **нощ**, но тарифата на единия превоз (такси) **зависи**. По тази причина **първата проверка** ще е именно дали е **ден или нощ**, за да стане ясно коя тарифа на таксито ще се **използва**. За целта **декларираме още една променлива**, в която ще пазим стойността на **тарифата на таксито**.

```
double taxiRate = 0;
```

За да изчислим **тарифата на таксито**, ще използваме проверка от типа **if-else** и чрез нея променливата за цената на таксито ще си присвои **стойност**.

```
if (dayOrNight == "day")
    taxiRate = 0.79;
else
    taxiRate = 0.90;
```

След като е направено и това, вече може да пристъпим към изчислението на са-
мата **цена за транспорта**. Ограниченията, които присъстват в условието на задача-
та, са относно **разстоянието**, което студента иска да пропътува. По тази причина,
ще построим **if-else** конструкция, с чиято помош ще открием **цената** за транс-
порта в зависимост от подадените километри.

```
if (distance < 20)
    price = 0.70 + distance * taxiRate;
else if (distance < 100)
    price = distance * 0.09;
else
    price = distance * 0.06;
```

Първо правим проверка дали километрите са **под 20**, тъй като от условието е
видно, че **под 20** километра студента би могъл да използва само **такси**. Ако усло-
вието на проверката е **вярно** (връща **true**), променливата, която е създадена, за
да пази стойността на цената на транспорта (**price**), ще си **присвои** съответната
стойност. Тази стойност е равна на **първоначалната такса**, която **събираме** с него-
вата **тарифа, умножена по разстоянието**, което студента трябва да измине.

Ако условието на променливата **не е вярно** (връща **false**), следващата стъпка е
програмата ни да провери дали километрите са **под 100**. Правим това, защото от
условието е видно, че в този диапазон може да се използва и **автобус** като транс-
порtnо средство. **Цената** за километър на автобуса е **по-ниска** от тази на таксито.
Следователно, ако резултата от проверката е **верен**, то в блок тялото на **else if**,
на променливата за цената на транспорта (**price**) трябва да присвоим **стойност**,
равна на резултата от **умножението** на **тарифата** на автобуса по **разстоянието**.

Ако и тази проверка **не даде true** като резултат, остава в тялото на **else** на про-
менливата за цена да присвоим **стойност**, равна на **резултата** от **умножението** на

разстоянието по тарифата на влака. Това се прави, тъй като влакът е **най-евтиния** вариант за транспорт при даденото разстояние.

Отпечатване на изходните данни

След като сме направили **проверките** за разстоянието и сме **изчислили цената** на **най-евтиния транспорт**, следва да я **отпечатаме**. В условието на задачата **няма** изисквания как да бъде форматиран резултата и по тази причина ще отпечатаме само **променливата**.

```
Console.WriteLine(price);
```

Тестване в Judge системата

Тествайте решението си: <https://judge.softuni.bg/Contests/Practice/Index/507#0>.

Задача: тръби в басейн

Басейн с **обем V** има **две тръби**, от които се пълни. Всяка тръба има определен **дебит** (литрите вода, минаващи през една тръба за един час). Работникът пуска тръбите едновременно и излиза за **N часа**. Напишете програма, която изкарва състоянието на басейна, **в момента**, когато работникът се върне.

Входни данни

От конзолата се четат **четири реда**:

- Първият ред съдържа числото **V** – обем на басейна в **литри** – цяло число в интервала **[1 ... 10000]**.
- Вторият ред съдържа числото **P1** – **дебит на първата тръба за час** – цяло число в интервала **[1 ... 5000]**.
- Третият ред съдържа числото **P2** – **дебит на втората тръба за час** – цяло число в интервала **[1 ... 5000]**.
- Четвъртият ред съдържа числото **H** – **часовете, в които работникът отсъства** – число с плаваща запетая в интервала **[1.0 ... 24.00]**.

Изходни данни

Да се отпечата на конзолата **едно от двете възможни състояния**:

- До колко се е запълнил басейнът и коя тръба с колко процента е допринесла. Всички проценти да се форматират до цяло число (без закръгляне).
 - "The pool is [x]% full. Pipe 1: [y]%. Pipe 2: [z]%"
- Ако басейнът се е препълнил – с колко литра е прелял за даденото време, число с плаваща запетая.
 - "For [x] hours the pool overflows with [y] liters."

Имайте предвид, че поради закръглянето до цяло число се губят данни и е нормално сборът на процентите да е 99%, а не 100%.

Примерен вход и изход

| Вход | Изход |
|------|---|
| 1000 | |
| 100 | |
| 120 | |
| 3 | The pool is 66% full. Pipe 1: 45%. Pipe2: 54%. |

| Вход | Изход |
|------|--|
| 100 | |
| 100 | |
| 100 | |
| 2.5 | For 2.5 hours the pool overflows with 400 liters. |

Насоки и подсказки

За да решим задачата, ще прочетем входа, ще извършим няколко проверки и изчисления и ще отпечатаме резултата.

Обработка на входните данни

От условието на задачата виждаме, че в програмата ни трябва да има **четири реда**, от които четем **входните данни**. Чрез първите **три** ще въвеждаме **цели числа** и по тази причина **променливите**, в които ще запазваме стойностите, ще бъдат от тип **int**. За **четвъртия** ред ни е казано, че ще бъде **число**, което е **с плаваща запетая**, затова и **променливата**, която ще използваме, е от тип **double**.

```
int volume = ...  
int pipe1 = ...  
int pipe2 = ...  
double hours = ...
```

Следващата ни стъпка е да **декларираме и инициализираме** променлива, в която ще изчислим с колко **литра** се е **напълнил** басейна за **времето**, в което работникът е **отсъствал**. Изчисленията ще направим като **съберем** стойностите на дебита на двете тръби и ги **умножим** по **часовете**, които са ни зададени като вход.

```
double water = ...
```

Извършване на проверки и обработка на изходните данни

След като вече имаме и **стойността на количеството** вода, което е минало през **тръбите**, следва стъпката, в която трябва да **сравним** това количество с обема на самия басейн.

Това ще направим с проста **if-else** проверка, в която условието ще е дали **количеството вода е по-малко от обема на басейна**. Ако проверката върне **true**, то трябва да разпечатаме един **ред**, който да съдържа в себе си **съотношението** между количеството **вода**, минало през **тръбите**, и **обема на басейна**, както и

съотношението на количеството вода от всяка една тръба спрямо обема на басейна.

Съотношението е нужно да бъде изразено в **проценти**, затова и всички изчисления до момента в този ред ще бъдат **умножени по 100**. Стойностите ще бъдат вмъкнати с **placeholder-и** и тъй като има условие **резултата в проценти** да се форматира до **две цифри** след десетичния знак **без закръгляне**, то за целта ще използваме метода **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);
}
```

Ако проверката обаче върне резултат **false**, то това означава, че **количеството вода е по-голямо** от **обемана басейна**, съответно той е **прелял**. Отново изхода трябва да е на **един ред**, но този път съдържа в себе си само **две стойности** – тази на **часовете**, в които работникът е отсъствал, и **количеството вода**, което е **разлика** между **влязлата вода** и **обема на басейна**.

Тестване в Judge системата

Тествайте решението си: <https://judge.softuni.bg/Contests/Practice/Index/507#1>.

Задача: поспаливата котка Том

Котката Том обича по цял ден да спи, за негово съжаление стопанинът му си играе с него винаги когато има свободно време. За да се наспи добре, **нормата за игра на Том е 30 000 минути в година**. Времето за игра на Том **зависи от почивните дни на стопанина му**:

- Когато е на **работа**, стопанинът му си играе с него **по 63 минути на ден**.
- Когато **почива**, стопанинът му си играе с него **по 127 минути на ден**.

Напишете програма, която въвежда броя **почивни дни** и отпечатва дали **Том може да се наспи добре** и колко е **разликата от нормата** за текущата година, като приемем че **годината има 365 дни**.

Пример: 20 почивни дни -> работните дни са 345 ($365 - 20 = 345$). Реалното време за игра е 24 275 минути ($345 * 63 + 20 * 127$). Разликата от нормата е 5 725 минути ($30\ 000 - 24\ 275 = 5\ 725$) или 95 часа и 25 минути.

Входни данни

Входът се чете от конзолата и се състои от едно цяло число – **броят почивни дни** в интервала [0...365].

Изходни данни

На конзолата трябва да се отпечатат **два реда**.

- Ако времето за игра на Том е **над нормата** за текущата година:
 - На първия ред отпечатайте: “Tom will run away”
 - На втория ред отпечатайте разликата от нормата във формат: “[H] hours and [M] minutes more for play”
- Ако времето за игра на Том е **под нормата** за текущата година:
 - На първия ред отпечатайте: “Tom sleeps well”
 - На втория ред отпечатайте разликата от нормата във формат: “[H] hours and [M] minutes less for play”

Примерен вход и изход

| Вход | Изход | Вход | Изход |
|------|---|------|--|
| 20 | Tom sleeps well 95 hours and 25 minutes less for play | 113 | Tom will run away 3 hours and 47 minutes for play |

Насоки и подсказки

За да решим задачата, ще прочетем входа, ще извършим няколко проверки и изчисления и ще отпечатаме резултата.

Обработка на входните данни и прилежащи изчисления

От условието на задачата виждаме, че **входните данни** ще бъдат прочетени само от **един ред**, който ще съдържа в себе си **едно цяло число** в интервала [0...365]. По тази причина ще използваме променлива от тип **int**.

```
var holidays =
```

За да решим задачата, **първо** трябва да изчислим колко **общо минути** стопанинът на Том си играе с него. От условието виждаме, че освен в **почивните дни**, поспаливатата котка трябва да си играе и в **работните** за стопанина му. Числото, което прочитаме от конзолата, е това на **почивните дни**.

Следващата ни стъпка е с помощта на това число да **изчислим** колко са **работните дни** на стопанина, тъй като без тях не можем да стигнем до **общото количество**

минути за игра. Щом общият брой на дните в годината е **365**, а броят на почивните дни е **X**, то това означава, че броят на работните дни е **365 – X**. Разликата ще запазим в нова променлива, която ще използваме **само** за тази **стойност**.

```
var workingDays =
```

След като вече имаме **количествата дни за игра**, то вече можем да **изчислим времето за игра** на Том в минути. Неговата **стойност е равна на резултата от умножението на работните дни по 63** минути (в условието е зададено, че в работни дни, времето за игра е 63 минути на ден) **събран с резултата от умножението на почивните дни по 127** минути (в условието е зададено, че в почивните дни, времето за игра е 127 минути на ден).

```
var totalPlayMinutes =
```

В условието на задачата за изхода виждаме, че ще трябва да **разпечатаме разликата** между двете стойности в **часове и минути**. За тази цел от **общото време за игра** ще **извадим** нормата от **30 000** минути и получената разлика ще **запишем в нова** променлива. След това тази променлива ще **разделим целочислено** на 60, за да получим **часовете**, а след това, за да открием колко са **минутите** ще използваме **модулно деление с оператора %**, като отново ще разделим променливата на разликата с 60.

Тук трябва да отбележим, че ако полученото количество **време игра** на Том е **помалко** от **30 000**, при **изваждането** на нормата от него ще получим **число с отрицателен знак**. За да **неутрализираме** знака в двете деления по-късно, ще използваме **метода Math.Abs(...)** при намирането на разликата.

```
var difference = Math.Abs();
var hours =
var minutes =
```

Извършване на проверки

Времето за игра вече е изчислено, което ни води до **следващата** стъпка – **сравняване на времето за игра** на Том с **нормата**, от която зависи дали котката може да се наспива добре. За целта ще използваме **if-else** проверка, като в **if** клавицата ще проверим дали **времето за игра** е **по-голямо** от **30 000** (нормата).

Обработка на изходните данни

Какъвто и **результат** да ни върне проверката, то трябва да разпечатаме колко е **разликата в часове и минути**. Това ще направим с **placeholder** и променливите, в които изчислихме стойностите на часовете и минутите, като форматирането ще е според условието за изход.

```
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);
}
```

Тестване в Judge системата

Тествайте решението си: <https://judge.softuni.bg/Contests/Practice/Index/507#2>.

Задача: реколта

От лозе с площ X квадратни метри се заделя 40% от реколтата за производство на вино. От 1 кв.м. лозе се изкарват Y килограма грозде. За 1 литър вино са нужни 2,5 кг. грозде. Желаното количество вино за продан е Z литра.

Напишете програма, която пресмята колко вино може да се произведе и дали това количество е достатъчно. Ако е достатъчно, остатъкът се разделя по равно между работниците на лозето.

Входни данни

Входът се чете от конзолата и се състои от **точно 4 реда**:

- 1-ви ред: X кв.м е лозето – цяло число в интервала [10 ... 5000].
- 2-ри ред: Y грозде за един кв.м. – реално число в интервала [0.00 ... 10.00].
- 3-ти ред: Z нужни литри вино – цяло число в интервала [10 ... 600].
- 4-ти ред: брой работници – цяло число в интервала [1 ... 20].

Изходни данни

На конзолата трябва да се отпечата следното:

- Ако произведеното вино е **по-малко** от нужното:
 - "It will be a tough winter! More {недостигащо вино} liters wine needed."
* Резултатът трябва да е закръглен към по-ниско цяло число.
- Ако произведеното вино е **повече** от нужното:
 - "Good harvest this year! Total wine: {общо вино} liters."
* Резултатът трябва да е закръглен към по-ниско цяло число.

- “{Оставащо вино} liters left -> {вино за 1 работник} liters per person.”
 * И двата резултата трябва да са закръглени към по-високото цяло число.

Примерен вход и изход

| Вход | Изход | Вход | Изход |
|----------------------|--|-------------------------|---|
| 650 2 175 3 | Good harvest this year! Total wine: 208 liters. 33 liters left -> 11 liters per person. | 1020 1.5 425 4 | It will be a tough winter! More 180 liters wine needed. |

Насоки и подсказки

За да решим задачата, ще прочетем входа, ще извършим няколко проверки и изчисления и ще отпечатаме резултата.

Обработка на входните данни и прилежащи изчисления

Първо трябва да проверим какви ще са **входните данни** и да изберем какви **променливи** ще използваме.

```
var vineyardArea = ...  
var grapePerSquare = ...  
var neededLiters = ...  
var workers = ...
```

За да решим задачата е нужно да **изчислим** колко **литра вино** ще получим на база **входните данни**. От условието на задачата виждаме, че за да **пресметнем** количеството **вино в литри**, трябва първо да разберем какво е **количество грозде в килограми**, което ще се получи от тази реколта. За тази цел ще **декларираме** една **променлива**, на която ще присвоим **стойност**, равна на **40%** от резултата от **умножението** на площта на лозето и количеството грозде, което се получава от 1 кв.м.

След като сме извършили тези пресмятания, сме готови да **пресметнем** и **количество вино в литри**, което ще се получи от тази реколта. За тази цел **декларираме** още една **променлива**, в която ще пазим това **количество**, а от условието стигаме до извода, че за да го пресметнем, е нужно да **разделим** количеството грозде в кг на 2.5.

```
var harvestPerVine = ...  
var vine = ...
```

Извършване на проверки и обработка на изходните данни

Вече направили нужните пресмятания и **следващата стъпка** е да проверим дали **получените литри вино** са **достатъчни**. За целта ще използваме **проста условна конструкция** от типа **if-else**, като в условието ще проверим дали **литрите вино** от реколтата са **повече от** или **равни на** **нужните литри**.

Ако проверката върне резултат **true**, от условието на задачата виждаме, че на **първия ред** трябва да разпечатаме **виното**, което сме получили от реколтата. За да спазим условието **тази стойност** да бъде закръглена до по-ниското **цяло число**, ще използваме метода **Math.Floor(...)** при разпечатването й чрез **placeholder**.

На втория ред има изискване да разпечатаме резултатите, като ги **закръглим към по-високото цяло число**, което ще направим с метода **Math.Ceiling(...)**. Стойностите, които трябва да разпечатаме, са на **оставащото количество вино** и **количеството вино**, което **се пада на един работник**. Оставащото количество вино е равно на **разликата** между **получените литри вино** и **нужните литри вино**. Стойността на това количество ще изчислим в нова променлива, която ще декларираме и инициализираме в **блок тялото** на **if**, **преди** разпечатването на първия ред. Количество вино, което **се полага на един работник**, ще изчислим като **оставащото вино го разделим на броя на работниците**.

Ето долу-горе как може да се извършват описаните изчисления:

```
if ( )
{
    var vineLeft = ...;
    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(...));
}
```

Ако проверката ни върне резултат **false** от условието на задачата виждаме, че трябва да **разпечатаме разликата** от **нужните литри** и **получените от тази реколта литри вино**. Има условие резултата да е **закръглен към по-ниското цяло число**, което ще направим с метода **Math.Floor...**

```
else
{
    Console.WriteLine("It will be a tough winter! More {0} liters wine needed.",
        Math.Floor(...));
}
```

Тестване в Judge системата

Тествайте решението си: <https://judge.softuni.bg/Contests/Practice/Index/507#3>.

Задача: фирма

Фирма получава заявка за изработването на проект, за който са необходими определен брой часове. Фирмата разполага с определен брой дни. През 10% от дните служителите са на обучение и не могат да работят по проекта. Един нормален работен ден във фирмата е 8 часа. Всеки служител може да работи по проекта в извънработно време по 2 часа на ден.

Часовете трябва да са закръглени към по-ниско цяло число (например → 6.98 часа се закръглят на 6 часа).

Напишете програма, която изчислява дали фирмата може да завърши проекта навреме и колко часа не достигат или остават.

Входни данни

Входът се чете от конзолата и съдържа точно 3 реда:

- На първия ред са необходимите часове – цяло число в интервала [0 ... 200 000].
- На втория ред са дните, с които фирмата разполага – цяло число в интервала [0 ... 20 000].
- На третия ред е броят на служителите, работещи извънредно – цяло число в интервала [0 ... 200].

Изходни данни

Да се отпечата на конзолата един ред:

- Ако времето е достатъчно:
 - "Yes!{оставащите часове} hours left."
- Ако времето НЕ Е достатъчно:
 - "Not enough time!{недостигащите часове} hours needed."

Примерен вход и изход

| Вход | Изход | Вход | Изход |
|--------------|-------------------|--------------|----------------------------------|
| 90 7 3 | Yes!2 hours left. | 99 3 1 | Not enough time!72 hours needed. |

Насоки и подсказки

За да решим задачата, ще прочетем входа, ще извършим няколко проверки и изчисления и ще отпечатаме резултата.

Обработка на входните данни

За решението на задачата е нужно **първо** да преценим какви **типове променливи** ще използваме за **входните данни**.

```
var projectHours = ...;
var availableDays = ...;
var overtimeWorkers = ...;
```

Помощни изчисления

Следващата стъпка е да изчислим **количеството на работните часове** като умножим работните дни по 8 (всеки ден се работи по 8 часа) и ги съберем с извънработното време. **Работните дни** са равни на **90% от дните**, с които фирмата разполага. **Извънработното време** е равно на резултата от умножението на броя на работещите извънредно служители с 2 (възможните часове извънработно време), като това също се умножава по броя на работните дни. От условието на задачата виждаме, че има условие **часовете да са закръглени към по-ниско цяло число**, кое-то ще направим с метода **Math.Floor(...)**.

```
var workDays = ...;
var overtime = ...;
var workHours = ...;
```

Извършване на проверки

След като сме направили изчисленията, които са ни нужни за да разберем стойността на **работните часове**, следва да направим проверка дали тези часове **достигат или остават допълнителни** такива.

Ако **времето е достатъчно**, разпечатваме резултата, който се изисква в условието на задачата, а именно разликата между **работните часове и необходимите часове** за завършване на проекта.

Ако **времето не е достатъчно**, разпечатваме допълнителните часове, които са нужни за завършване на проекта и са равни на разликата между **часовете за проекта и работните часове**.

Ето и приблизително как можем да реализираме отпечатването:

```
if ( ... )
{
    Console.WriteLine("Yes! {0} hours left.", ... );
}
else
{
    Console.WriteLine("Not enough time! {0} hours needed.", ... );
```

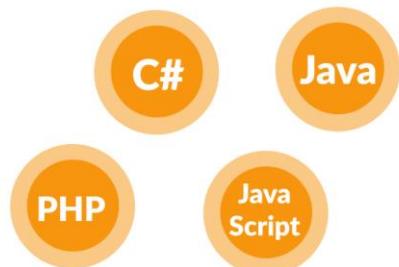
Тестване в Judge системата

Тествайте решението си: <https://judge.softuni.bg/Contests/Practice/Index/507#4>.

Качествено образование,
професия и работа за

Софтуерни инженери

- ✓ Безплатен старт за начинаещи - **присъствено и онлайн**
- ✓ Избор измежду **най-търсените професии** в софтуерната индустрия
- ✓ Съдействие за **кариерен старт**
- ✓ Отлични преподаватели, ментори и студентска общност



Софтуни предоставя съвременно иновативно образование за програмиране, ИТ и дигитални умения на хиляди млади хора. Програмата "**Софтуерен университет**" изгражда истински **професионалисти в света на програмирането**.

Включете се в цялостната програма по софтуерно инженерство на Софтуни, за да усвоите **най-търсените съвременни софтуерни технологии** чрез модерна учебна методика и с много практически задачи и проекти. **Учебният план** е разработен съвместно с ИТ фирмите и изгражда най-търсените от тях умения.

Софтуни работи пряко с компаниите от софтуерната индустрия, съдейтайки на своите студенти за реализацията им като успешни софтуерни инженери.

ПЪТЯТ НА СТУДЕНТА В СОФТУНИ



1 - 1.5 години



1.5 - 2 години



Programming Basics

Кариерен Старт

Дипломиране

70/100 credits

80/100/150 credits

Кандидатствай

softuni.bg/apply

Глава 4.1. По-сложни проверки

В настоящата глава ще разгледаме **вложените проверки** в езика C#, чрез които нашата програма може да съдържа **условни конструкции**, в които има **вложени други условни конструкции**. Наричаме ги "вложени", защото **поставяме if конструкция в друга if конструкция**. Ще разгледаме и **по-сложни логически условия** с подходящи примери.

Видео

Гледайте видео-урок по тази глава: <https://youtube.com/watch?v=z8XxYIyesz0>.

Вложени проверки

Доста често програмната логика налага използването на **if** или **if-else** конструкции, които се съдържат една в друга. Те биват наричани **вложени if** или **if-else** конструкции. Както се подразбира от названието "вложени", това са **if** или **if-else** конструкции, които са поставени в други **if** или **else** конструкции.

```
if (condition1)
{
    if (condition2)
    {
        // тяло;
    }
    else
    {
        // тяло;
    }
}
```

Влагането на повече от три условни конструкции една в друга не се счита за добра практика и трябва да се избягва, най-вече чрез оптимизиране на структурата / алгоритъма на кода или чрез използването на друг вид условна конструкция, които ще разгледаме по-надолу в тази глава.

Пример: обръщение според възраст и пол

Според въведени **възраст** (десетично число) и **пол** (**m** / **f**) да се отпечата обръщение:

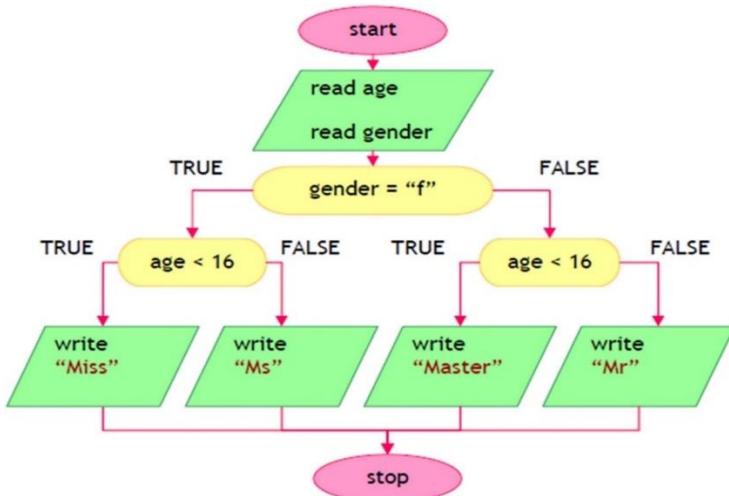
- "Mr." – мъж (пол "m") на 16 или повече години.
- "Master" – момче (пол "m") под 16 години.
- "Ms." – жена (пол "f") на 16 или повече години.
- "Miss" – момиче (пол "f") под 16 години.

Примерен вход и изход

| Вход | Изход | Вход | Изход | Вход | Изход | Вход | Изход |
|---------|-------|---------|-------|---------|-------|-----------|--------|
| 12 f | Miss | 17 m | Mr. | 25 f | Ms. | 13.5 m | Master |

Решение

Можем да забележим, че **изходът** на програмата зависи от **няколко неща**. Първо трябва да проверим какъв **пол** е въведен и **после** да проверим **възрастта**. Съответно ще използваме **няколко if-else** блока. Тези блокове ще бъдат **вложени**, т.е. от **результатата** на първия ще определим кои от **другите** да изпълним.



След прочитане на входните данни от конзолата ще тряба да се изпълни следната примерна логика:

```

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.");
}
  
```

Тестване на решението:

Тествайте решението си: <https://judge.softuni.bg/Contests/Practice/Index/508#0>.

Пример: квартално магазинче

Предприемчив българин отваря по едно квартално магазинче в няколко града с различни цени за следните продукти:

| продукт / град | 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 |

По даден град (стринг), продукт (стринг) и количество (десетично число) да се пресметне цената.

Примерен вход и изход

| Вход | Изход | Вход | Изход | Вход | Изход | Вход | Изход |
|--------|-------|---------|-------|-------|-------|---------|-------|
| coffee | | peanuts | | beer | | water | |
| Varna | 0.9 | Plovdiv | 1.5 | Sofia | 7.2 | Plovdiv | 2.1 |
| 2 | | 1 | | 6 | | 3 | |

Решение

Прехвърляме всички букви в долн регистър с функцията `.ToLower()`, за да сравняваме продукти и градове без значение от малки / главни букви.

```
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 ...
```

Тестване на решението:

Тествайте решението си: <https://judge.softuni.bg/Contests/Practice/Index/508#1>.

По-сложни проверки

Нека разгледаме как можем да правим по-сложни логически проверки. Може да използваме логическо "И" (`&&`), логическо "ИЛИ" (`||`), логическо **отрицание** (`!`) и скоби (`()`).

Логическо "И"

Както видяхме, в някои задачи се налага да правим **много проверки наведнъж**. Но какво става, когато за да изпълним някакъв код, трябва да бъдат изпълнени **повече условия и не искаем** да правим **отрицание** (`else`) за всяко едно от тях? Вариантът с вложените **if блокове** е валиден, но кодът би изглеждал много **неподреден** и със сигурност – **труден** за четене и поддръжка.

Логическо "И" (оператор `&&`) означава няколко условия да са **изпълнени едновременно**. В сила е следната таблица на истинност:

| a | b | a && b |
|-------|-------|--------|
| true | true | true |
| true | false | false |
| false | true | false |
| false | false | false |

Как работи операторът `&&` ?

Операторът `&&` приема **няколко булеви** (условни) израза, които имат стойност **true** или **false**, и ни връща **един** булев израз като **результат**. Използването му **вместо** редица вложени **if блокове** прави кода **по-четлив, подреден и лесен** за поддръжка. Но как **работи**, когато поставим **няколко** условия едно след друго? Както видяхме по-горе, логическото "И" връща **true**, само когато приема като **аргументи изрази** със стойност **true**. Съответно, когато имаме **последователност** от аргументи, логическото "И" **проверява** или докато **свършат** аргументите, или докато не **срещне** аргумент със стойност **false**.

Пример:

```
bool a = true;
bool b = true;
bool c = false;
bool d = true;
bool result = a && b && c && d; // false (като d не се проверява)
```

Програмата ще се изпълни по **следния** начин: **започва** проверката от **a**, прочита я и отчита, че има стойност **true**, след което **проверява b**. След като е **отчела**, че **a** и **b** връщат стойност **true**, **проверява следващия** аргумент. Стига до **c** и отчита, че резултатът е вече **false**. Проверката на **d** се **прескача**, защото и целият израз бива изчислен като **false** и няма нужда изчислението да продължава.

```
if (x >= x1 && x <= x2 && y >= y1 && y <= y2)
```

Пример: точка в правоъгълник

Проверка дали точка $\{x, y\}$ се намира вътре в правоъгълника $\{x_1, y_1\} - \{x_2, y_2\}$. Входните данни се четат от конзолата и се състоят от 6 реда: десетичните числа x_1, x_2, y_2, x и y (като се гарантира, че $x_1 < x_2$ и $y_1 < y_2$).

Примерен вход и изход

| Вход | Изход | Визуализация |
|-------------------------------|--------|--------------|
| 2 -3 12 3 8 -1 | Inside | |

Решение

Една точка е вътрешна за даден многоъгълник, ако едновременно са изпълнени следните четири условия:

- Точката е надясно от лявата страна на правоъгълника.
- Точката е наляво от дясната страна на правоъгълника.
- Точката е надолу от горната страна на правоъгълника.
- Точката е нагоре от долната страна на правоъгълника.

```
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");
}

```

Тестване на решението

Тествайте решението си: <https://judge.softuni.bg/Contests/Practice/Index/508#2>.

Логическо "ИЛИ"

Логическо "ИЛИ" (оператор `||`) означава да е **изпълнено поне едно** измежду няколко условия. Подобно на оператора `&&`, логическото "ИЛИ" приема няколко аргумента от булев (условен) тип и връща `true` или `false`. Лесно можем да се досетим, че **получаваме** като стойност `true`, винаги когато поне **един** от аргументите има стойност `true`. Типичен пример за логиката на този оператор е следният:

В училище учителят казва: "Иван или Петър да измият дъската". За да бъде изпълнено това условие (дъската да бъде измита), е възможно само Иван да я измие, само Петър да я измие или и двамата да го направят.

| a | b | <code>a b</code> |
|-------|-------|---------------------|
| true | true | true |
| true | false | true |
| false | true | true |
| false | false | false |

Как работи операторът `||`?

Вече научихме какво **представлява** логическото "ИЛИ". Но как всъщност се реализира? Както при логическото "И", програмата **проверява** от ляво на дясно **аргументите**, които са зададени. За да получим `true` от израза, е необходимо **само един** аргумент да има стойност `true`, съответно проверката **продължава** докато се срещне аргумент с **такава** стойност или докато **не свършат** аргументите.

Ето един **пример** за оператора `||` в действие:

```

bool a = false;
bool b = true;
bool c = false;
bool d = true;

```

```
bool result = a || b || c || d;
// true (като с и d не се проверяват)
```

Програмата **проверява** **a**, отчита, че има стойност **false** и продължава. Стигайки до **b**, отчита, че има стойност **true** и целият **израз** получава стойност **true**, **без** да се проверява **c** и **d**, защото техните стойности **не биха променили** резултата на израза.

Пример: плод или зеленчук

Нека проверим дали даден **продукт** е **плод** или **зеленчук**. Плодовете "fruit" са **banana**, **apple**, **kiwi**, **cherry**, **lemon** и **grapes**. Зеленчуците "vegetable" са **tomato**, **cucumber**, **pepper** и **carrot**. Всички останали са "**unknown**".

Примерен вход и изход

| Вход | Изход |
|--------|-----------|
| banana | fruit |
| tomato | vegetable |
| java | unknown |

Решение

Можем да използваме няколко условни проверки с логическо "ИЛИ" (`||`):

```
string s = Console.ReadLine().ToLower();
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");
}
```

Тестване на решението:

Тествайте решението си: <https://judge.softuni.bg/Contests/Practice/Index/508#3>.

Логическо отрицание

Логическо отрицание (оператор **!**) означава да **не е изпълнено** дадено условие.

| | |
|------|-------|
| a | !a |
| true | false |

Операторът **!** приема като **аргумент** булева променлива и **обръща** стойността ѝ.

Пример: невалидно число

Дадено число е **валидно**, ако е в диапазона [100...200] или е 0. Да се направи проверка за **невалидно** число.

Примерен вход и изход

| Вход | Изход |
|------|--------------|
| 75 | invalid |
| 150 | (няма изход) |
| 220 | invalid |

Решение

```
var inRange = (num >= 100 && num <= 200) || num == 0;
if (!inRange)
    Console.WriteLine("invalid");
```

Тестване на решението

Тествайте решението си: <https://judge.softuni.bg/Contests/Practice/Index/508#4>.

Операторът скоби ()

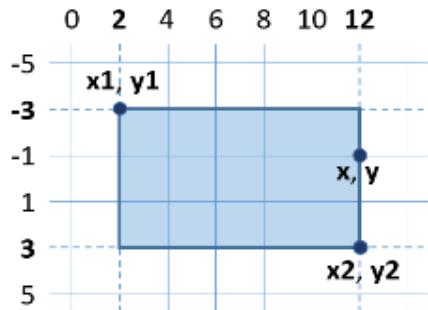
Както останалите оператори в програмирането, така и операторите **&&** и **||** имат приоритет, като в случая **&&** е с по-голям приоритет от **||**. Операторът **()** служи за **промяна на приоритета на операторите** и се изчислява пръв, също както в математиката. Използването на скоби също така придава по-добра четимост на кода и се счита за добра практика.

По-сложни логически условия

Понякога условията може да са доста сложни, така че да изискват дълъг булев израз или поредица от проверки. Да разгледаме няколко такива примера.

Пример: точка върху страна на правоъгълник

Да се напише програма, която проверява дали точка $\{x, y\}$ се намира **върху** някоя от страните на правоъгълник $\{x_1, y_1\} - \{x_2, y_2\}$. Входните данни се четат от конзолата и се състоят от 6 реда: десетичните числа x_1, y_1, x_2, y_2, x и y (като се гарантира, че $x_1 < x_2$ и $y_1 < y_2$). Да се отпечата "Border" (точката лежи на някоя от страните) или "Inside / Outside" (в противен случай).



Примерен вход и изход

| Вход | Изход | Вход | Изход |
|------|--------|------|------------------|
| 2 | | 2 | |
| -3 | | -3 | |
| 12 | Border | 12 | |
| 3 | | 3 | Inside / Outside |
| 12 | | 8 | |
| -1 | | -1 | |

Решение

Точка лежи върху някоя от страните на правоъгълник, ако:

- x съвпада с x_1 или x_2 и същевременно y е между y_1 и y_2 или
- y съвпада с y_1 или y_2 и същевременно x е между x_1 и x_2

```
if (((x == x1 || x == x2) && (y >= y1) && (y <= y2)) ||
    ((y == y1 || y == y2) && (x >= x1) && (x <= x2)))
{
    Console.WriteLine("Border");
}
```

Предходната проверка може да се опрости по начина, даден по-долу.

Вторият начин с допълнителните булеви променливи е по-дълъг, но е много по-разбираем от първия, нали? Препоръчваме ви когато пищете булеви условия, да ги правите **лесни за четене и разбиране**, а не кратки. Ако се налага, ползвайте допълнителни променливи със смислени имена. Имената на булевите променливи трябва да подсказват каква стойност се съхранява в тях.

```
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");
}
```

Остава да допишете кода, за да отпечатва “Inside / Outside”, ако точката не е върху някоя от страните на правоъгълника.

Тестване на решението

След като допишете решението, може да го тествате тук:

<https://judge.softuni.bg/Contests/Practice/Index/508#5>.

Пример: магазин за плодове

Магазин за плодове в **работни дни** продава на следните **цени**:

| Плод | Цена |
|------------|------|
| banana | 2.50 |
| apple | 1.20 |
| orange | 0.85 |
| grapefruit | 1.45 |
| kiwi | 2.70 |
| pineapple | 5.50 |
| grapes | 3.85 |

В почивни дни цените на същите плодове са **по-високи**:

| Плод | Цена |
|------------|------|
| banana | 2.70 |
| apple | 1.25 |
| orange | 0.90 |
| grapefruit | 1.60 |
| kiwi | 3.00 |
| pineapple | 5.60 |
| grapes | 4.20 |

Напишете програма, която **чете** от конзолата **плод** (banana / apple / ...), **ден от седмицата** (Monday / Tuesday / ...) и **количество** (десетично число) и пресмята **цената** според цените от таблиците по-горе. Резултатът да се отпечата **закръглен** с 2 цифри след десетичния знак. При **невалиден ден** от седмицата или **невалидно име** на плод да се отпечата “error”.

Примерен вход и изход

| Вход | Изход |
|--------|-------|
| orange | |
| Sunday | |
| 3 | 2.70 |

| Вход | Изход |
|--------|-------|
| kiwi | |
| Monday | |
| 2.5 | 6.75 |

| Вход | Изход |
|----------|-------|
| grapes | |
| Saturday | |
| 0.5 | 2.10 |

| Вход | Изход |
|--------|-------|
| tomato | |
| Monday | |
| 0.5 | error |

Решение

Можем да изградим следната логика от вложени проверки:

```

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 ...
}

```

Тестване на решението

Тествайте решението си: <https://judge.softuni.bg/Contests/Practice/Index/508#6>.

Пример: търговски комисионни

Фирма дава следните комисионни на търговците си според града, в който работят и обема на продажбите s:

| Град | $0 \leq s \leq 500$ | $500 < s \leq 1000$ | $1000 < s \leq 10000$ | $s > 10000$ |
|---------|---------------------|---------------------|-----------------------|-------------|
| Sofia | 5% | 7% | 8% | 12% |
| Varna | 4.5% | 7.5% | 10% | 13% |
| Plovdiv | 5.5% | 8% | 12% | 14.5% |

Напишете **програма**, която чете име на **град** (стринг) и обем на **продажбите** (десетично число) и изчислява размера на комисионната. Резултатът да се изведе закръглен с **2 десетични цифри след десетичния знак**. При **невалиден град или обем на продажбите** (отрицателно число) да се отпечата "**error**".

Примерен вход и изход

| Вход | Изход | Вход | Изход | Вход | Изход |
|---------------|--------|-------------------|-------|------------------|-------|
| Sofia 1500 | 120.00 | Plovdiv 499.99 | 27.50 | Kaspichan -50 | error |

Решение

При прочитането на входа можем да обърнем града в малки букви (с функцията **.ToLower()**). Първоначално задаваме комисионната да е **-1**. Тя ще бъде променена, ако градът и ценовият диапазон бъдат намерени в таблицата с комисионните.

За да изчислим комисионната според града и обема на продажбите се нуждаем от няколко вложени **if** проверки, както е в примерния код по-долу:

```
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");
```

Тестване на решението

Тествайте решението си: <https://judge.softuni.bg/Contests/Practice/Index/508#7>.



Добра практика е да използваме **блокове**, които **заграждаме** с къдрави скоби **{ }** след **if** и **else**. Също така, препоръчително е при писане да **отместваме** кода **след if и else** с една табулация **навътре**, за да направим кода по-лесно четим.

Условна конструкция switch-case

Конструкцията **switch-case** работи като поредица **if-else** блокове. Когато работата на програмата ни зависи от стойността на **една променлива**, вместо да правим последователни проверки с **if-else** блокове, можем да **използваме** условната конструкция **switch**. Тя се използва за **избор измежду списък с възможности**. Конструкцията сравнява дадена стойност с определени константи и в зависимост от резултата предприема действие.

Променливата, която искаме да **сравняваме**, поставяме в **скобите след оператора switch** и се нарича "селектор". Тук типът трябва да е **сравним** (числа, стрингове). Последователно започва **сравняването** с всяка една **стойност**, която се **намира** след **case етикетите**. При съвпадение започва изпълнението на кода от съответното място и продължава, докато стигне оператора **break**. В някои програмни езици (като C и C++) **break** може да се изпуска, за да се изпълнява код от друга **case** конструкция, докато не стигне до въпросния оператор. В C# обаче, наличието на **break** е **задължителен** за **всеки case**, който съдържа изпълнение на програмна логика. При липса на **съвпадение**, се изпълнява **default** конструкцията, ако такава съществува.

```

switch (селектор)
{
    case стойност1:
        конструкция;
        break;
    case стойност2:
        конструкция;
        break;
    case стойност3:
        конструкция;
        break;
    ...
    default:
        конструкция;
        break;
}

```

Пример: ден от седмицата

Нека напишем програма, която принтира **дения от седмицата** (на английски) според въведеното число (1...7) или "Error!", ако е подаден невалиден ден.

Примерен вход и изход

| Вход | Изход |
|------|--------|
| 1 | Monday |
| 7 | Sunday |
| -1 | Error! |

Вместо да правим поредица от **if-else-if...** конструкции, ще е по-лесно ако имаме конструкция за групова проверка на множество условия на веднъж.

Решение

Ако използваме **switch-case** конструкцията, можем да направим серия проверки в много по-компактен и разбираем вид, отколкото с **if-else-if...** конструкции.

Ето едно примерно решение на задачата:

```

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;
}

```



Добра практика е на първо място да поставяме онези **case** случаи, които обработват най-често случилите се ситуации, а **case** конструкциите, обработващи по-рядко възникващи ситуации, да оставим в края преди **default** конструкцията.

Друга добра практика е да подреждаме **case** етикетите в нарастващ ред, без значение дали са целочислени или символни.

Тестване на решението

Тествайте решението си: <https://judge.softuni.bg/Contests/Practice/Index/508#8>.

Множество етикети в switch-case

В C# имаме възможността да използваме множество **case** етикети, когато те трябва да изпълняват един и същи код. При този начин на записване, когато програмата ни намери съвпадение, ще изпълни следващия срещнат код, тъй като след съответния **case** етикет липсва код за изпълнение и **break** оператор.

```

switch (селектор)
{
    case стойност1:
    case стойност2:
    case стойност3:
        конструкция;
        break;
    case стойност4:
    case стойност5:
        конструкция;
        break;
    ...
    default:
        конструкция;
        break;
}

```

Пример: вид животно

Напишете програма, която принтира вида на животно според името му:

- dog → mammal
- crocodile, tortoise, snake → reptile
- others → unknown

Примерен вход и изход

| Вход | Изход |
|----------|---------|
| tortoise | reptile |

| Вход | Изход |
|------|--------|
| dog | mammal |

| Вход | Изход |
|----------|---------|
| elephant | unknown |

Решение

Можем да решим задачата чрез **switch-case** проверки с множество етикети по следния начин:

```
switch (animal)
{
    case "dog": Console.WriteLine("mammal"); break;
    case "crocodile":
    case "tortoise":
    case "snake": Console.WriteLine("reptile"); break;
    default: Console.WriteLine("unknown"); break;
}
```

Тестване на решението

Тествайте решението си: <https://judge.softuni.bg/Contests/Practice/Index/508#9>.

Какво научихме от тази глава?

Да си припомним новите конструкции, които научихме в тази глава.

Вложени проверки

```
if (condition1)
{
    if (condition2)
        // тяло;
    else
        // тяло;
}
```

По-сложни проверки с &&, ||, ! и ()

```
if ((x == left || x == right) && y >= top && y <= bottom)
    Console.WriteLine(...);
```

Switch-case проверки

```
switch (селектор)
{
    case стойност1:
        конструкция;
        break;
    case стойност2:
    case стойност3:
        конструкция;
        break;
    ...
    default:
        конструкция;
        break;
}
```

Упражнения: по-сложни проверки

Нека сега да упражним работата с по-сложни проверки. Да решим няколко практически задачи.

Задача: кино

В една кинозала столовете са наредени в **правоъгълна** форма в **r** реда и **c** колони. Има три вида прожекции с билети на **различни цени**:

- **Premiere** – премиерна прожекция, на цена **12.00** лева.
- **Normal** – стандартна прожекция, на цена **7.50** лева.
- **Discount** – прожекция за деца, ученици и студенти на намалена цена от **5.00** лева.

Напишете програма, която въвежда **тип прожекция** (стринг), брой **редове** и брой **колони** в залата (цели числа) и изчислява **общите приходи** от билети при **пълна зала**. Резултатът да се отпечата във формат като в примерите по-долу – с 2 цифри след десетичния знак.

Примерен вход и изход

| Вход | Изход |
|----------|--------------|
| Premiere | |
| 10 | 1440.00 leva |
| 12 | |

| Вход | Изход |
|--------|--------------|
| Normal | |
| 21 | 2047.50 leva |
| 13 | |

Насоки и подсказки

При прочитането на входа можем да обърнем типа на прожекцията в малки букви (с функцията `.ToLower()`). Създаваме и инициализираме променлива, която ще ни съхранява изчислените приходи. В друга променлива пресмятаме пълния капацитет на залата. Използваме **switch-case** условна конструкция, за да изчислим прихода в зависимост от вида на прожекцията и отпечатваме резултата на конзолата в зададения формат (потърсете нужната C# функционалност в интернет).

Примерен код (части от кода са замъглени с цел да се стимулира самостоятелно мислене и решение):

```
string type = Console.ReadLine().ToLower();
int rows = [REDACTED];
int columns = [REDACTED];

int full = rows * columns;
double income = [REDACTED];

switch (type)
{
    case "premiere":
        income = full * 35.00;
        break;
    case "normal":
        income = full * 11.00;
        break;
    case "discount":
        income = full * 5.00;
        break;
}

Console.WriteLine($"[REDACTED] leva");
```

Тестване на решението

Тествайте решението си: <https://judge.softuni.bg/Contests/Practice/Index/508#10>.

Задача: волейбол

Влади е студент, живее в София и си ходи от време на време до родния град. Той е много запален по волейбола, но е зает през работните дни и играе **волейбол**

само през уикендите и в празничните дни. Влади играе в София всяка събота, когато не е на работа и не си пътува до родния град, както и в 2/3 от празничните дни. Той пътува до родния си град h пъти в годината, където играе волейбол със старите си приятели в неделя. Влади не е на работа 3/4 от уикендите, в които е в София. Отделно, през високосните години Влади играе с 15% повече волейбол от нормалното. Приемаме, че годината има точно 48 уикенда, подходящи за волейбол. Напишете програма, която изчислява колко пъти Влади е играл волейбол през годината. **Закръглете резултата** надолу до най-близкото цяло число (напр. 2.15 -> 2; 9.95 -> 9).

Входните данни се четат от конзолата:

- Първият ред съдържа думата “**leap**” (високосна година) или “**normal**” (нормална година с 365 дни).
- Вторият ред съдържа цялото число **p** – брой празници в годината (които не са събота или неделя).
- Третият ред съдържа цялото число **h** – брой уикенди, в които Влади си пътува до родния град.

Примерен вход и изход

| Вход | Изход | Вход | Изход | Вход | Изход | Вход | Изход |
|----------------|-------|------------------|-------|-------------------|-------|----------------|-------|
| leap 5 2 | 45 | normal 3 2 | 38 | normal 11 6 | 44 | leap 0 1 | 41 |
| | | | | | | | |

Насоки и подсказки

Стандартно прочитаме входните данни от конзолата като за избягване на грешки при въвеждане обръщаме текста в малки букви с функцията **.ToLower()**. Последователно пресмятаме **уикендите прекарани в София**, времето за игра в София и **общото време за игра**. Накрая проверяваме дали годината е **високосна**, правим допълнителни изчисления при необходимост и извеждаме резултата на конзолата **закръглен надолу** до най-близкото **цяло число** (потърсете **C#** клас с такава функционалност в интернет).

Примерен код (части от кода са замъглени с цел да се стимулира самостоятелно мислене и решение):

```
string year = Console.ReadLine().ToLower();
int holidays = 0;
int weekendsHome = 0;
int sofiaWeekends = 0;
double playSofia = 0.0;
double playTotal = 0.0;

int SofiaWeekends = 0;
double playSofia = 0.0;
double playTotal = 0.0;
```

```

if (user.Equals("One"))
{
    playTotal = Math.Floor(playTotal) * (15 / 1000 + playTotal);
}
else if (user.Equals("Two"))
{
    playTotal = Math.Floor(playTotal));
}
Console.WriteLine(playTotal);

```

Тестване на решението

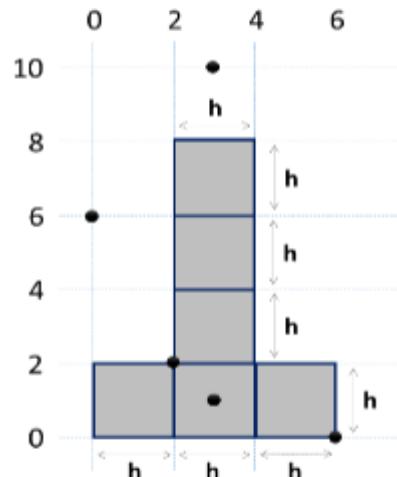
Тествайте решението си: <https://judge.softuni.bg/Contests/Practice/Index/508#11>.

Задача: * точка във фигурата

Фигура се състои от 6 блокчета с размер $h \times h$, разположени като на фигурата. Долният ляв ъгъл на сградата е на позиция $\{0, 0\}$. Горният десен ъгъл на фигурата е на позиция $\{2 \cdot h, 4 \cdot h\}$. На фигурата координатите са дадени при $h = 2$.

Да се напише програма, която въвежда цяло число h и координатите на дадена точка $\{x, y\}$ (цели числа) и отпечатва дали точката е вътре във фигурата (**inside**), вън от фигурата (**outside**) или на някоя от стените на фигурата (**border**).

Примерен вход и изход



| Вход | Изход | Вход | Изход | Вход | Изход | Вход | Изход |
|------|---------|------|--------|------|--------|------|--------|
| 2 | | 2 | | 2 | | 2 | |
| 3 | outside | 3 | inside | 2 | border | 6 | border |
| 10 | | 1 | | 2 | | 0 | |

| Вход | Изход | Вход | Изход | Вход | Изход | Вход | Изход |
|------|---------|------|---------|------|--------|------|---------|
| 2 | | 15 | | 15 | | 15 | |
| 0 | outside | 13 | outside | 29 | inside | 37 | outside |
| 6 | | 55 | | 37 | | 18 | |

Насоки и подсказки

Примерна логика за решаване на задачата (не е единствената правилна):

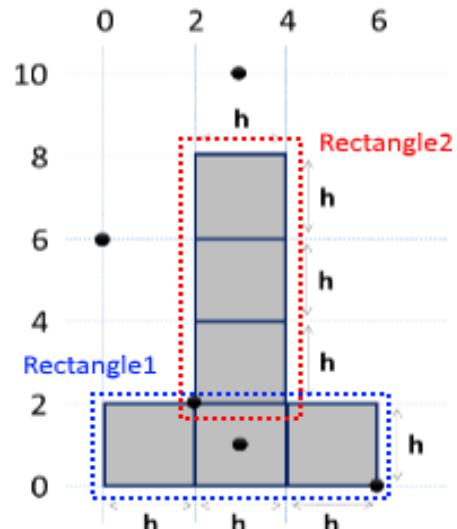
- Може да разделим фигурата на **два правоъгълника** с обща стена (вж. чертежа).
- Една точка е **външна (outside)** за фигурата, когато е едновременно **извън** двета правоъгълника.
- Една точка е **вътрешна (inside)** за фигурата, ако е вътре в някой от правоъгълниците (изключвайки стените им) или лежи върху общата им стена.
- В **противен случай** точката лежи на стената на правоъгълника (**border**).

Следва примерен код, в който части от кода са замъглени с цел да се стимулира самостоятелно мислене и решение:

```
int h = ...;  
int x = ...;  
int y = ...;
```

След прочитане на размерите на фигурата се извършват описаните проверки за положението на точката спрямо двета правоъгълника и спрямо тях се отпечатва и крайният резултат:

```
bool outRectangle1 = ...;  
bool outRectangle2 = ...;  
  
bool inRectangle1 = ...;  
bool inRectangle2 = ...;  
  
bool commonBorder = ...;  
  
if (!outRectangle1 && !outRectangle2)  
{  
    Console.WriteLine("inside");  
}  
else if (!inRectangle1 || !inRectangle2 || commonBorder)  
{  
    Console.WriteLine("on border");  
}
```



```

else
{
    Console.WriteLine("border");
}

```

Тестване на решението

Тествайте решението си в online judge системата: <https://judge.softuni.bg/Contests/Practice/Index/508#12>.

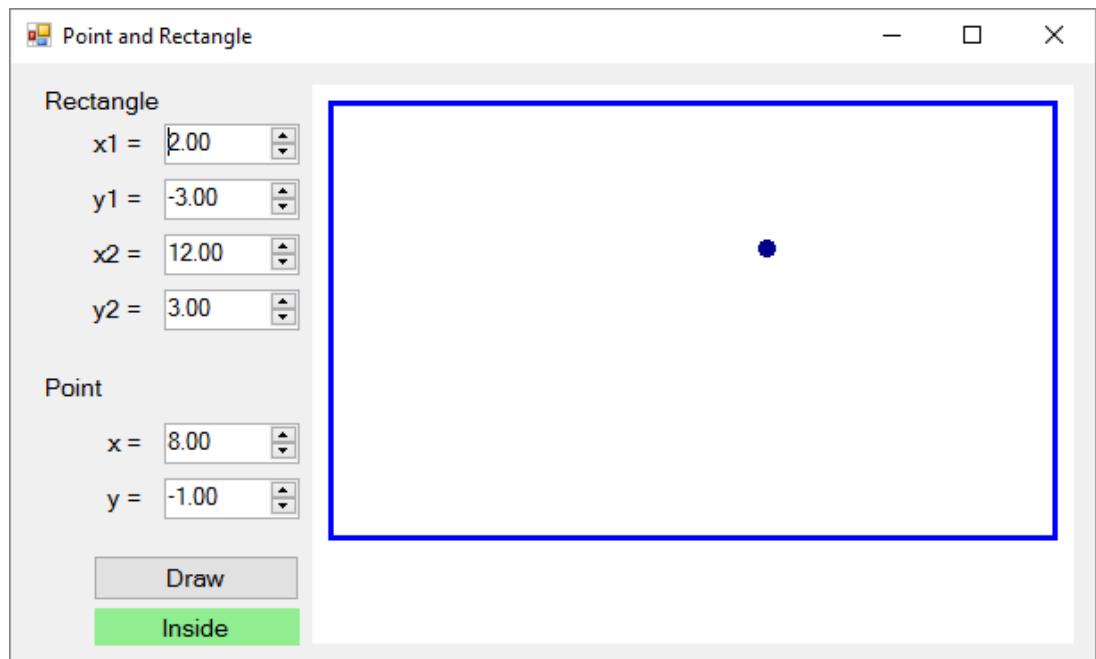
Тази задача беше малко по-трудна от предходните, но пък стимулира **алгоритмичното ви мислене**, а то е по-важно от писането на код.

Упражнения: GUI с по-сложни проверки

В тази глава научихме как можем да правим **проверки с нетривиални условия**. Нека сега приложим тези знания, за да създадем нещо интересно: настолно приложение, което визуализира точка и правоъгълник. Това е прекрасна визуализация за една от задачите от упражненията.

Задача: * Точка и правоъгълник – графично (GUI) приложение

Задачата, която си поставяме е да се разработи графично (GUI) приложение за **визуализация на точка и правоъгълник**. Приложението трябва да изглежда приблизително по следния начин:

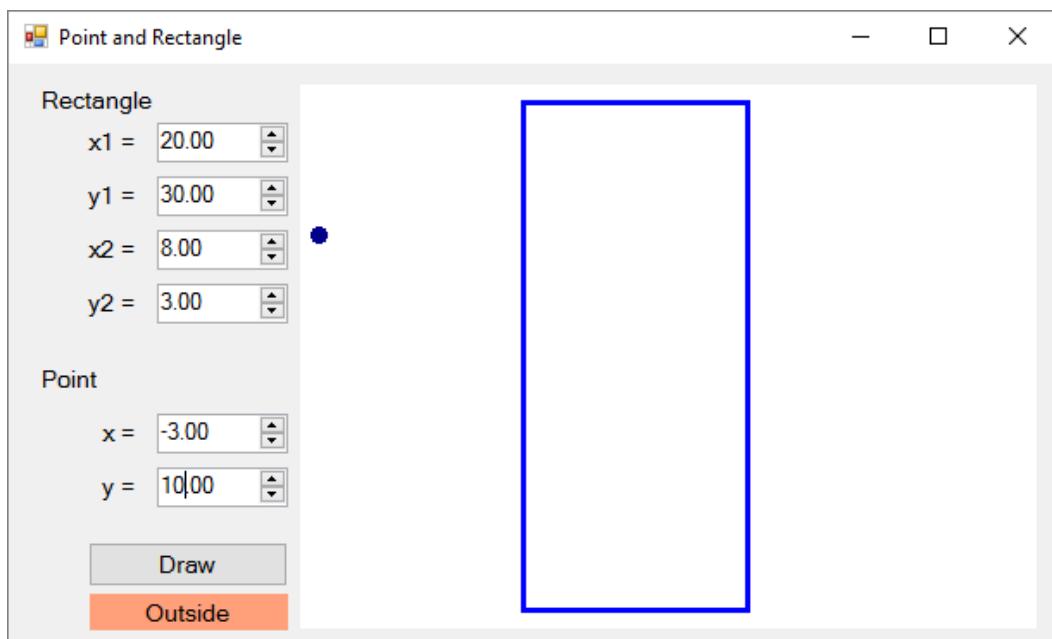
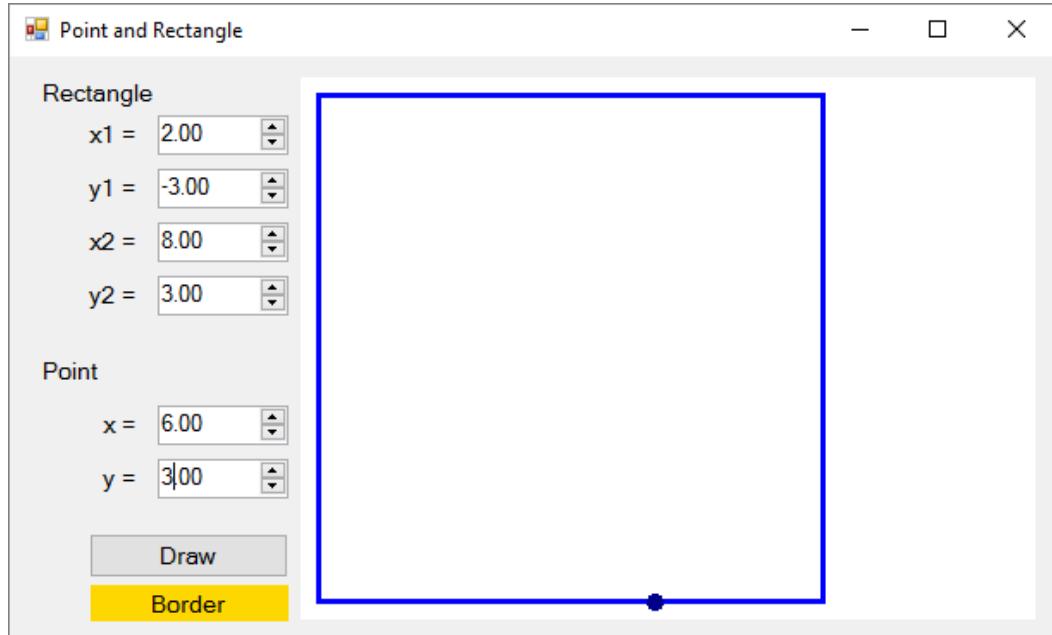


От контролите вляво се задават координатите на **два от ъглите** на правоъгълник (десетични числа) и координатите на **точка**. Приложението **визуализира** графично

правоъгълника и точката и изписва дали точката е **вътре** в правоъгълника (**Inside**), **вън** от него (**Outside**) или на някоя от стените му (**Border**).

Приложението **премества и мащабира** координатите на правоъгълника и точката, за да бъдат максимално големи, но да се събират в полето за визуализация в дясната страна на приложението, без да се появява скролер.

Ето още два примера за визуализация на точка съответно върху контура на правоъгълника и извън правоъгълника:

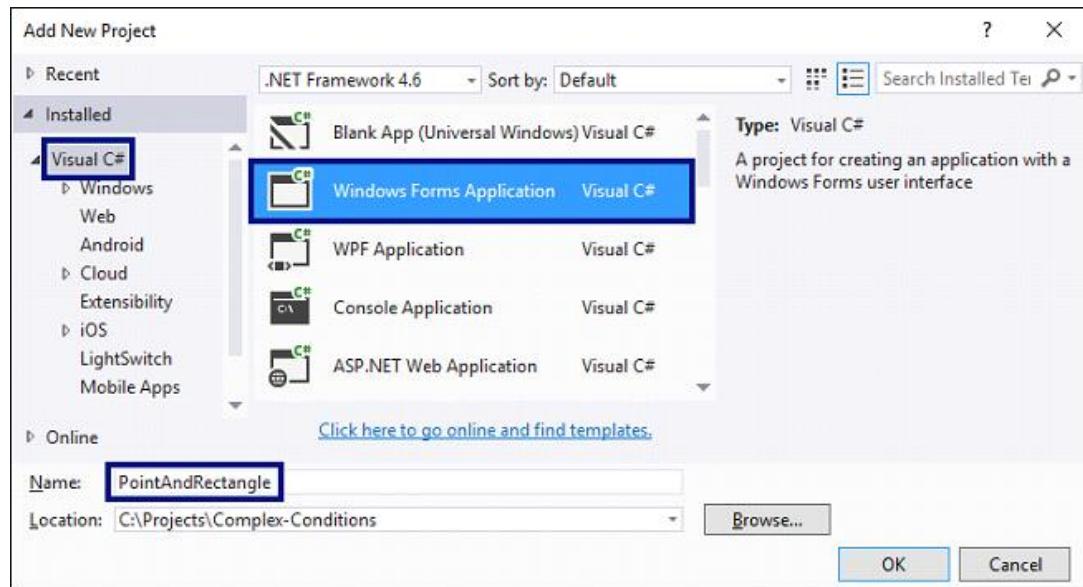




Внимание: това приложение е значително **по-сложно** от предходните графични приложения, които разработвахме до сега, защото изисква ползване на функции за чертане и нетривиални изчисления за преоразмеряване и преместване на правоъгълник и точка.

Да се захващаме за работа. Първо ще създадем **нов проект**, после ще сложим в него контролите (**UI controls**) за въвеждане на координати и останалите визуални елементи и ще настроим някои техни свойства (**Properties**), след което ще напишем кода, който се изпълнява при натискане на **[Draw]** бутона. Накрая ще стартираме и тестваме приложението с няколко различни примера, за да хванем всички по-интересни ситуации по тази задача.

Първата стъпка е да създадем **Windows Forms** проект. Можем да използваме **същия Solution**, в който са останалите проекти с решенията на задачите от тази глава на книгата. Във Visual Studio създаваме нов проект **Windows Forms Application** с подходящо име, например "**Point-and-Rectangle**".



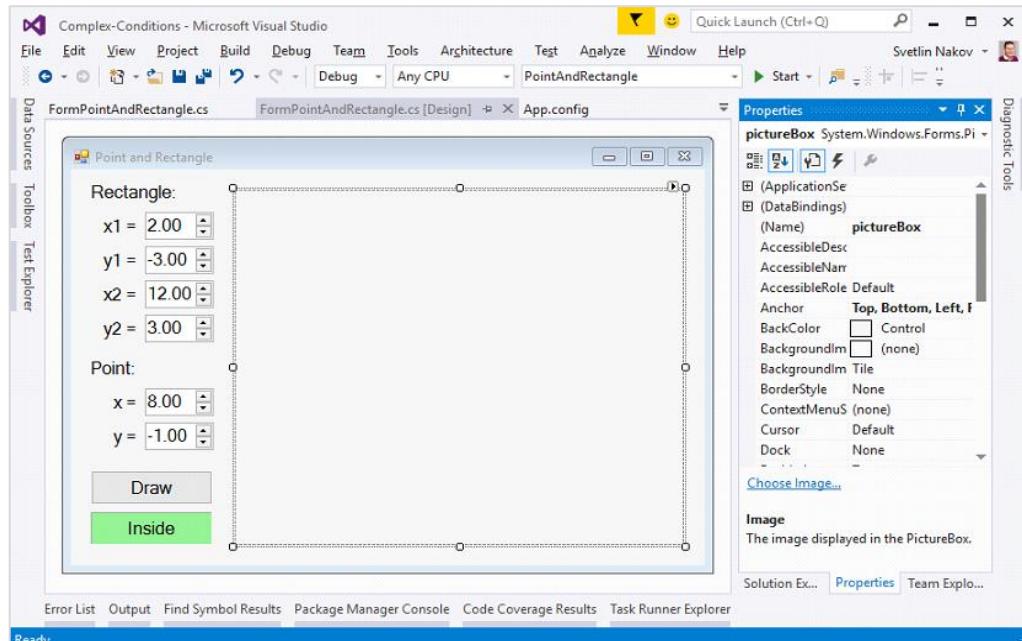
Нареждаме контролите във формата, както е показано на фигурата по-долу:

- 6 кутийки за въвеждане на число (**NumericUpDown**).
- Надписи (**Label**) пред всяка кутийка за въвеждане на число.
- Бутон (**Button**) за изчертаване на правоъгълника и точката.
- Текстов блок за резултата (**Label**).

Нагласяме **размерите** и **свойствата** на контролите, за да изглеждат приблизително като на картинката по-долу.

Задаваме следните препоръчителни **настройки** на контролите:

- За главната форма (**Form**), която съдържа всички контроли:
 - (name) = **FormPointAndRectangle**
 - Text** = **Point and Rectangle**
 - Font.Size** = **12**
 - Size** = **700, 410**
 - MinimumSize** = **500, 400**
 - FormBorderStyle** = **FixedSingle**



- За полетата за въвеждане на число (**NumericUpDown**):
 - (name) = **numericUpDownX1; numericUpDownY1; numericUpDownX2; numericUpDownY2; numericUpDownX; numericUpDownY**
 - Value** = **2; -3; 12; 3; 8; -1**
 - Minimum** = **-100000**
 - Maximum** = **100000**
 - DecimalPlaces** = **2**
- За бутона (**Button**) за визуализация на правоъгълника и точката:
 - (name) = **buttonDraw**
 - Text** = **Draw**
- За текстовия блок за резултата (**Label**):
 - (name) = **labelLocation**
 - AutoSize** = **False**

- **BackColor** = **PaleGreen**;
- **TextAlign** = **MiddleCenter**
- За полето с чертежа (**PictureBox**):
 - (name) = **pictureBox**
 - **Anchor** = **Top, Bottom, Left, Right**

Следва да хванем следните **събития**, за да напишем C# кода, който ще се изпълни при настъпването им:

- Събитие **Click** на бутона **buttonDraw** (извиква се при натискане на бутона).
- Събитие **ValueChanged** на контролите за въвеждане на числа **numericUpDownX1**, **numericUpDownY1**, **numericUpDownX2**, **numericUpDownY2**, **numericUpDownX** и **numericUpDownY** (извиква се при промяна на стойността в контролата за въвеждане на число).
- Събитие **Load** на формата **FormPointAndRectangle** (извиква се при стартиране на приложението, преди да се появи главната форма на екрана).
- Събитие **Resize** на формата **FormPointAndRectangle** (извиква се при промяна на размера на главната формата).

Всички изброени по-горе събития ще изпълняват едно и също действие – **Draw()**, което ще визуализира правоъгълника и точката и ще показва дали тя е вътре, вън или на някоя от страните. Кодът трябва да изглежда така:

```

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 event handlers
numericUpDownY1_ValueChanged,
numericUpDownX2_ValueChanged,
numericUpDownY2_ValueChanged,
numericUpDownX_ValueChanged and
numericUpDownY_ValueChanged */

private void Draw()
{
    // TODO: implement this a bit later ...
}

```

Нека започнем от по-лесната част: **печат на информация къде е точката спрямо правоъгълника** (Inside, Outside или Border). Кодът трябва да изглежда така:

```

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;
    }
}

```

Горният код взима координатите на правоъгълника и точките и проверява дали точката е **вътре**, **вън** или **на страната** на правоъгълника. При визуализацията на резултата се сменя и цвета на фона на текстовия блок, който го съдържа.

Помислете как **да допишете** недовършените (нарочно) условия в **if проверките!** Кодът по-горе **нарочно не се компилира**, защото целта му е да помислите как и защо работи и да **допишете сами липсващите части**.

Остава да се имплементира най-сложната част: визуализация на правоъгълника и точката в контролата **pictureBox** с преоразмеряване. Може да си помогнем с **кода по-долу**, който прави малко изчисления и рисува син правоъгълник и тъмно-синьо кръгче (точката) според зададените във формата координати. За съжаление сложността на кода надхвърля изучавания до момента материал и е сложно да се обясни в детайли как точно работи. Оставени са коментари за ориентация. Това е пълната версия на действието **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));
}
```

В горния код се срещат доста **преобразувания на типове**, защото се работи с различни типове числа (десетични числа, реални числа и цели числа) и понякога се изисква да се преминава между тях.

Накрая **компилираме кода**. Ако има грешки, ги отстраняваме. Най-вероятната **причина** за грешка е несъответстващо име на някоя от контролите или ако **сте написали кода на неправилно място**.

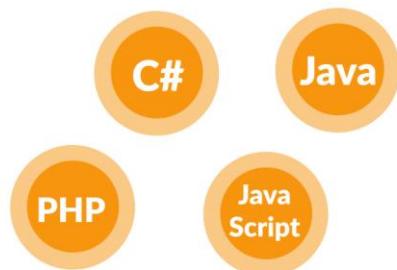
Стартираме приложението и го **тестваме**. Въвеждаме различни данни, за да видим дали се държи коректно.

Ако имате проблеми с примерния проект по-горе, **гледайте видеото** в началото на тази глава. Там приложението е направено на живо стъпка по стъпка с много обяснения. Или питайте във **форума на СофтУни**: <https://softuni.bg/forum>.

Качествено образование,
професия и работа за

Софтуерни инженери

- ✓ Безплатен старт за начинаещи - **присъствено и онлайн**
- ✓ Избор измежду **най-търсените професии** в софтуерната индустрия
- ✓ Съдействие за **кариерен старт**
- ✓ Отлични преподаватели, ментори и студентска общност



Софтуни предоставя съвременно иновативно образование за програмиране, ИТ и дигитални умения на хиляди млади хора. Програмата "**Софтуерен университет**" изгражда истински **професионалисти в света на програмирането**.

Включете се в цялостната програма по софтуерно инженерство на Софтуни, за да усвояте **най-търсените съвременни софтуерни технологии** чрез модерна учебна методика и с много практически задачи и проекти. **Учебният план** е разработен съвместно с ИТ фирмите и изгражда най-търсените от тях умения.

Софтуни работи пряко с компаниите от софтуерната индустрия, съдейтайки на своите студенти за реализацията им като успешни софтуерни инженери.

ПЪТЯТ НА СТУДЕНТА В СОФТУНИ



1 - 1.5 години



1.5 - 2 години



Programming Basics

Кариерен Старт

Дипломиране

70/100 credits

80/100/150 credits

Кандидатствай

softuni.bg/apply

Глава 4.2. По-сложни проверки – изпитни задачи

В предходната глава се запознахме с **вложените условни конструкции** в езика C#. Чрез тях програмната логика в дадена програма може да бъде представена посредством **if конструкции**, които се съдържат една в друга. Разгледахме и условната конструкция **switch-case**, която позволява избор между списък от възможности. Следва да упражним и затвърдим наученото досега, като разгледаме няколко по-сложни задачи, давани на изпити. Преди да преминем към задачите, ще си припомним условните конструкции:

Вложени проверки

```
if (condition1)
{
    if (condition2)
        // тяло;
    else
        // тяло;
}
```



Запомнете, че не е добра практика да пишете **дълбоко вложени условни конструкции** (с ниво на влагане повече от три). Избягвайте влагане на повече от три условни конструкции една в друга. Това усложнява кода и затруднява неговото четене и разбиране.

Switch-case проверки

Когато работата на програмата ни зависи от стойността на една променлива, вместо да правим последователни проверки с множество **if-else** блокове, можем да използваме условната конструкция **switch-case**.

```
switch (селектор)
{
    case стойност1:
        конструкция;
        break;
    case стойност2:
        конструкция;
        break;
    default:
        конструкция;
        break;
}
```

Конструкцията се състои от:

- Селектор – израз, който се изчислява до някаква конкретна стойност. Типът на селектора може да бъде **цяло число**, **string** или **enum**.
- Множество **case** етикети с команди след тях, завършващи с **break**.

Изпитни задачи

Сега, след като си припомнихме как се използват условни конструкции и как се влагат една в друга условни конструкции, за реализиране на по-сложни проверки и програмна логика, нека решим няколко изпитни задачи.

Задача: навреме за изпит

Студент трябва да отиде **на изпит в определен час** (например в 9:30 часа). Той идва в изпитната зала в даден **час на пристигане** (например 9:40). Счита се, че студентът е дошъл **навреме**, ако е пристигнал в часа на изпита или до половин час **преди това**. Ако е пристигнал **по-рано** повече от 30 минути, той е **подранил**. Ако е дошъл **след часа на изпита**, той е **закъснял**.

Напишете програма, която въвежда време на изпит и време на пристигане и отпечатва дали студентът е дошъл **навреме**, дали е **подранил** или е **закъснял**, както и с колко часа или минути е подранил или закъснял.

Входни данни

От конзолата се четат **четири цели числа** (по едно на ред):

- Първият ред съдържа **час на изпита** – цяло число от 0 до 23.
- Вторият ред съдържа **минута на изпита** – цяло число от 0 до 59.
- Третият ред съдържа **час на пристигане** – цяло число от 0 до 23.
- Четвъртият ред съдържа **минута на пристигане** – цяло число от 0 до 59.

Изходни данни

На първия ред отпечатайте:

- "Late", ако студентът пристига **по-късно** от часа на изпита.
- "On time", ако студентът пристига **точно** в часа на изпита или до 30 минути по-рано.
- "Early", ако студентът пристига повече от 30 минути **преди** часа на изпита.

Ако студентът пристига с поне минута разлика от часа на изпита, отпечатайте на следващия ред:

- "**mm minutes before the start**" за идване по-рано с по-малко от час.
- "**hh:mm hours before the start**" за подраняване с 1 час или повече. Минутите винаги печатайте с 2 цифри, например "1:05".

- "mm minutes after the start" за закъснение под час.
- "hh:mm hours after the start" за закъснение от 1 час или повече. Минутите винаги печатайте с 2 цифри, например "1:03".

Примерен вход и изход

| Вход | Изход | Вход | Изход | Вход | Изход |
|--------------------|------------------------------------|----------------------|--------------------------------------|--------------------|---|
| 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 |

| Вход | Изход | Вход | Изход | Вход | Изход |
|---------------------|------------------------------------|----------------------|---------------------------------------|----------------------|---------|
| 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 |

Насоки и подсказки



Препоръчително е да прочетете няколко пъти заданието на дадена задача, като си водите записи и си скицирате примерите, докато разсъждавате над тях, преди да започнете писането на код.

Обработка на входните данни

Съгласно заданието очакваме да ни бъдат подадени **четири** поредни реда с различни **цели числа**. Разглеждайки дадените параметри можем да се спрем на типа **int**, тъй като той удовлетворява очакваните ни стойности. Едновременно **четем** входа и **парсваме** стринговата стойност към избрания от нас целочислен тип.

```
var examHours = int.Parse(Console.ReadLine());
var examMinutes = int.Parse(Console.ReadLine());
var arrivalHours = int.Parse(Console.ReadLine());
var arrivalMinutes = int.Parse(Console.ReadLine());
```

Разглеждайки очаквания изход можем да създадем променливи, които да съдържат различните видове изходни данни, с цел да избегнем използването на т.нар. "magic strings" в кода.

```
string late = "Late";
string onTime = "On time";
string early = "Early";
```

Изчисления

След като прочетохме входа, можем да започнем да разписваме логиката за изчисление на резултата. Нека първо да изчислим **началния час** на изпита **в минути** за по-лесно и точно сравнение.

```
int examTime = (examHours * 60) + examMinutes;
```

Нека изчислим по същата логика и времето на пристигане на студента.

```
int arrivalTime =
    (arrivalHours * 60) + arrivalMinutes;
```

Остава ни да пресметнем разликата в двете времена, за да можем да определим кога и с какво време спрямо изпита е пристигнал студентът.

```
int totalMinutesDifference = arrivalTime - examTime;
```

Следващата ни стъпка е да направим необходимите **проверки и изчисления**, като накрая ще изведем резултата от тях. Нека разделим изхода на **две** части.

- Първо да покажем кога е пристигнал студентът – дали е **подранил**, **закъснял** или е пристигнал **навреме**. За целта ще се спрем на **if-else** конструкция.
- След това ще покажем **времевата разлика**, ако студентът пристигне в **различно време** от началния час на изпита.

С цел да спестим една допълнителна проверка (**else**), можем по подразбиране да приемем, че студентът е закъснял.

След което, съгласно условието, проверяваме дали разликата във времената е **повече от 30 минути**. Ако това е така, приемаме, че е **подранил**. Ако не влезем в първото условие, то следва да проверим само дали **разликата е по-малка или равна на нула (<= 0)**, с което проверяваме условието, студентът да е дошъл в рамките на от **0 до 30 минути** преди изпита.

При всички останали случаи приемаме, че студентът е **закъснял**, което сме направили по подразбиране, и не е нужна допълнителна проверка.

```
string studentArrival = late;
if (totalMinutesDifference < -30)
{
    studentArrival = early;
}
else if (totalMinutesDifference <= 30)
{
    studentArrival = onTime;
}
```

За финал ни остава да разберем и покажем с **каква разлика от времето на изпита е пристигнал**, както и дали тази разлика показва време на пристигане **преди или след изпита**.

Правим проверка дали разликата ни е **над** един час, за да изпишем съответно часове и минути в желания по задание **формат**, или е **под** един час, за да покажем **само минути** като формат и описание.

Остава да направим още една проверка – дали времето на пристигане на студента е **преди** или **след** началото на изпита.

```
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";
    }
}
```

Отпечатване на резултата

Накрая остава да изведем резултата на конзолата. По задание, ако студентът е дошъл точно на време (без нито една минута разлика), не трябва да изваждаме втори резултат. Затова правим следната проверка:

```
Console.WriteLine(studentArrival);
if (!string.IsNullOrEmpty(result))
{
    Console.WriteLine(result);
}
```

Реално за целите на задачата извеждането на резултата **на конзолата** може да бъде направен и в по-ранен етап – още при самите изчисления. Това като цяло не е много добра практика. **Защо?**

Нека разгледаме идеята, че кодът ни не е 10 реда, а 100 или 1000! Някой ден ще се наложи извеждането на резултата да не бъде в конзолата, а да бъде записан във **файл** или показан на **уеб приложение**. Тогава на колко места в кода ще трябва да бъдат нанесени корекции поради тази смяна? И дали няма да пропуснем някое място?



Винаги си мислете за кода с логическите изчисления, като за отделна част от, различна от обработката на входните и изходните данни. Той трябва да може да работи без значение как му се подават данните и къде ще трябва да бъде показан резултатът.

Тестване в Judge системата

Тествайте решението си: <https://judge.softuni.bg/Contests/Practice/Index/509#0>.

Задача: пътешествие

Странно, но повечето хора си планиват от рано почивката. Млад програмист разполага с **определен бюджет** и свободно време в даден **сезон**.

Напишете програма, която да приема **на входа бюджета и сезона**, а **на изхода** да изкарва **къде ще почива** програмистът и **колко ще похарчи**.

Бюджетът определя дестинацията, а сезонът определя колко от бюджета ще бъде изхарчен. Ако е **лято**, ще почива на **къмпинг**, а **зимата** – в **хотел**. Ако е в **Европа**, независимо от сезона, ще почива в **хотел**. Всеки **къмпинг** или **хотел**, според **дестинацията**, има **собствена цена**, която отговаря на даден процент от бюджета:

- При **100 лв.** или **по-малко** – някъде в **България**.
 - **Лято** – **30%** от бюджета.
 - **Зима** – **70%** от бюджета.
- При **1000 лв.** или **по малко** – някъде на **Балканите**.
 - **Лято** – **40%** от бюджета.
 - **Зима** – **80%** от бюджета
- При **повече от 1000 лв.** – някъде из **Европа**.

- При пътуване из Европа, независимо от сезона, ще похарчи **90%** от бюджета.

Входни данни

Входът се чете от конзолата и се състои от **два реда**:

- На първия ред получаваме бюджета – реално число в интервал [10.00...5000.00].
- На втория ред – **един** от двата възможни сезона: "summer" или "winter".

Изходни данни

На конзолата трябва да се отпечатат **два реда**.

- На първи ред – "Somewhere in {дестинация}" между "Bulgaria", "Balkans" и "Europe".
- На втори ред – "{Вид почивка} – {Похарчена сума}".
 - Потребителят може да е между "Camp" и "Hotel".
 - Сумата трябва да е закръглена с точност до втория символ след десетичния знак.

Примерен вход и изход

| Вход | Изход |
|--------------|---------------------------------------|
| 50 summer | Somewhere in Bulgaria Camp – 15.00 |

| Вход | Изход |
|--------------|--|
| 75 winter | Somewhere in Bulgaria Hotel – 52.50 |

| Вход | Изход |
|---------------|---------------------------------------|
| 312 summer | Somewhere in Balkans Camp – 124.80 |

| Вход | Изход |
|----------------|--|
| 1500 summer | Somewhere in Europe Hotel – 1350.00 |

Насоки и подсказки

Типично, както и при другите задачи, можем да разделим решението на няколко части: четене на входните данни, изчисления, отпечатване на резултата.

Обработка на входните данни

Прочитайки внимателно условието разбираме, че очакваме **два** реда с входни данни. Първият параметър е **реално число**, за което е добре да изберем подходящ тип на променливата. За по-голяма точност в изчисленията ще се спрем на **decimal** като тип за бюджета, а за сезона – **string**.

```
decimal budget = decimal.Parse(Console.ReadLine());
string season = Console.ReadLine();
```



Винаги преценявайте какъв **тип стойност** се подава при входните данни, както и към какъв тип трябва да бъдат конвертирани тези данни, за да работят правилно създадените от вас програмни конструкции!

Пример: когато в задачата е необходимо да направите парични изчисления, използвайте **decimal** за по-добра точност.

Изчисления

Нека си създадем и инициализираме нужните за логиката и изчисленията променливи.

```
string destinationResult = string.Empty;
string holidayInformation = string.Empty;
decimal moneySpent = 0.00M;
```

Подобно на примера в предната задача, можем да инициализираме променливите с някои от изходните резултати – с цел спестяване на допълнително инициализиране.

Разглеждайки отново условието на задачата забелязваме, че основното разпределение за това къде ще почиваме се определя от **стойността на подадения бюджет**, т.е. основната ни логика се разделя на два случая:

- Ако бюджетът е **по-малък** от дадена стойност.
- Ако е **по-малък** от друга стойност, или е **повече** от дадена гранична стойност.

След това е необходимо да направим проверка за стойността на **подадения сезон**. Спрямо нея ще определим какъв процент от бюджета ще бъде похарчен, както и къде ще почива програмистът – в **хотел** или на **къмпинг**.

Пример за един от възможните подходи за решение е:

```
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);
}

```

Знаете ли, че спрямо това как си подредим логическата схема (в какъв ред ще обхождаме граничните стойности), ще имаме повече или по-малко проверки в условията. **Помислете защо!**

Винаги можем да инициализираме дадена стойност на параметъра и след това да направим само една проверка. Това ни спестява една логическа стъпка.

Например следният блок:

```

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);
    }
}

```

може да бъде съкратен до този вид:

```

destinationResult = "Bulgaria";
moneySpent = 0.70M * budget;
holidayInformation =
    string.Format("Hotel - {0:F2}", moneySpent);

if (season.Equals("summer"))
{
    moneySpent = 0.30M * budget;
    holidayInformation =
        string.Format("Camp - {0:F2}", moneySpent);
}

```

Отпечатване на резултата

Остава ни да покажем изчисления резултат на конзолата:

```

Console.WriteLine("Somewhere in {0}", destinationResult);
Console.WriteLine(holidayInformation);

```

Тестване в Judge системата

Тествайте решението си: <https://judge.softuni.bg/Contests/Practice/Index/509#1>.

Задача: операции между числа

Напишете програма, която чете две цели числа ($n1$ и $n2$) и оператор, с който да се извърши дадена математическа операция с тях. Възможните операции са: събиране (+), изваждане (-), умножение (*), деление (/) и модулно деление (%). При събиране, изваждане и умножение на конзолата трябва да се отпечата резултата и дали той е четен или нечетен. При обикновено деление – единствено резултата, а при модулно деление – остатъка. Трябва да се има предвид, че делителят може да е равен на нула (= 0), а на нула не се дели. В този случай трябва да се отпечата специално съобщение.

Входни данни

От конзолата се прочитат 3 реда:

- $N1$ – цяло число в интервала [0...40 000].
- $N2$ – цяло число в интервала [0...40 000].
- Оператор – един символ измежду: "+", "-", "*", "/", "%".

Изходни данни

Да се отпечата на конзолата един ред:

- Ако операцията е събиране, изваждане или умножение:
 - " $\{N1\} \{оператор\} \{N2\} = \{резултат\}$ – {even/odd}".
- Ако операцията е деление:
 - " $\{N1\} / \{N2\} = \{резултат\}$ " – резултатът е форматиран до втория символ след десетичния знак.
- Ако операцията е модулно деление:
 - " $\{N1\} \% \{N2\} = \{остатък\}$ ".
- В случай на деление на 0 (нула):
 - "Cannot divide $\{N1\}$ by zero".

Примерен вход и изход

| Вход | Изход |
|------|------------------|
| 10 | |
| 1 | 10 – 1 = 9 – odd |
| - | |

| Вход | Изход |
|------|------------------|
| 7 | |
| 3 | |
| * | 7 * 3 = 21 – odd |

| Вход | Изход |
|------|---------------------|
| 10 | |
| 12 | |
| + | 10 + 12 = 22 – even |

| Вход | Изход |
|------|------------------|
| 7 | |
| 3 | |
| * | 7 * 3 = 21 – odd |

| Вход | Изход |
|------|--------------------|
| 123 | |
| 12 | |
| / | $123 / 12 = 10.25$ |

| Вход | Изход |
|------|---------------|
| 10 | |
| 3 | |
| % | $10 \% 3 = 1$ |

Насоки и подсказки

Задачата не е сложна, но има доста редове код за писане.

Обработка на входните данни

След прочитане на условието разбираме, че очакваме **три** реда с входни данни. На първите **два** реда ни се подават **цели числа** (в указания от заданието диапазон), а на третия – **аритметичен символ**.

```
decimal n1 = decimal.Parse(Console.ReadLine());
decimal n2 = decimal.Parse(Console.ReadLine());
string nOperator = Console.ReadLine();
```

Изчисления

Нека си създадем и инициализираме нужните за логиката и изчисленията променливи. В едната ще пазим **резултата от изчисленията**, а другата ще използваме за **крайния изход** на програмата.

```
decimal result = 0.00M;
string output = string.Empty;
```

Прочитайки внимателно условието разбираме, че има случаи, в които не трябва да правим **никакви** изчисления, а просто да изведем резултат.

Следователно първо може да проверим дали второто число е **0** (нула), както и да ли операцията е **деление** или **модулно деление**, след което да инициализираме резултата.

```
if (n2 == 0 && (nOperator.Equals("/") || nOperator.Equals("%")))
{
    output = string.Format("Cannot divide {0} by zero", n1);
}
```

Нека сложим резултата като стойност при инициализацията на **output** параметъра. По този начин може да направим само **една проверка** – дали е необходимо да **преизчислим** и **заменим** този резултат.

Спрямо това кой подход изберем, следващата ни проверка ще бъде или обикновен **else** или единичен **if**. В тялото на тази проверка, с допълнителни проверки за начина на изчисление на резултата спрямо подадения оператор, можем да разделим логиката спрямо **структурата** на очаквания **резултат**.

От условието можем да видим, че за **събиране (+)**, **изваждане (-)** или **умножение (*)** очакваният резултат има еднаква структура: "**{n1} {оператор} {n2} = {резултат}**" – **{even/odd}**", докато за **деление (/)** и за **модулно деление (%)** резултатът има различна структура.

```
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);
}
```

Завършваме с проверките за събиране, изваждане и умножение:

```
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");
}
```

При кратки и ясни проверки, както в горния пример за четно и нечетно число, е възможно да се използва **тернарен оператор**. Нека разгледаме възможната проверка **c** и **без** тернарен оператор.

Без използване на тернарен оператор кодът е по-дълъг, но се чете лесно:

```
string numberIs = string.Empty;
if (result % 2 == 0)
{
    numberIs = "even";
}
else
{
    numberIs = "odd";
}
```

С използване на тернарен оператор кодът е много по-кратък, но може да изисква допълнителни усилия, за да бъде прочетен и разбран като логика:

```
string numberIs = result % 2 == 0 ? "even" : "odd";
```

Отпечатване на резултата

Накрая ни остава да покажем изчисления резултат на конзолата:

```
Console.WriteLine(output);
```

Тестване в Judge системата

Тествайте решението си: <https://judge.softuni.bg/Contests/Practice/Index/509#2>.

Задача: билети за мач

Група запалянковци решили да си закупят **билети за Евро 2016**. Цената на билета се определя спрямо **две** категории:

- VIP – 499.99 лева.
- Normal – 249.99 лева.

Запалянковците **имат определен бюджет**, а броят на хората в групата определя какъв процент от бюджета трябва да се задели за транспорт:

- От 1 до 4 – 75% от бюджета.
- От 5 до 9 – 60% от бюджета.
- От 10 до 24 – 50% от бюджета.
- От 25 до 49 – 40% от бюджета.
- 50 или повече – 25% от бюджета.

Напишете програма, която да пресмята дали с останалите пари от бюджета могат да си купят билети за избраната категория, както и колко пари ще им останат или ще са им нужни.

Входни данни

Входът се чете от конзолата и съдържа точно 3 реда:

- На първия ред е **бюджетът** – реално число в интервала [1 000.00 ... 1 000 000.00].
- На втория ред е **категорията** – "VIP" или "Normal".
- На третия ред е **броят на хората** в групата – цяло число в интервала [1 ... 200].

Изходни данни

Да се отпечатва на конзолата **един ред**:

- Ако бюджетът е достатъчен:
 - "Yes! You have {N} leva left." – където N са останалите пари на групата.
- Ако бюджетът НЕ Е достатъчен:
 - "Not enough money! You need {M} leva." – където M е сумата, която не достига.

Сумите трябва да са форматирани с точност до два символа след десетичния знак.

Примерен вход и изход

| Вход | Изход | Обяснения |
|---------------------|----------------------------------|---|
| 1000 Normal 1 | Yes! You have 0.01 leva left. | 1 човек : 75% от бюджета отиват за транспорт. Остават: $1000 - 750 = 250$. Категория Normal: билетът струва $249.99 * 1 = 249.99$ $249.99 < 250$: остават му $250 - 249.99 = 0.01$ |

| Вход | Изход | Обяснения |
|--------------------|---|--|
| 30000 VIP 49 | Not enough money! You need 6499.51 leva. | 49 човека: 40% от бюджета отиват за транспорт. Остават: $30000 - 12000 = 18000$. Категория VIP: билетът струва $499.99 * 49 = 24499.51$ $24499.51 < 18000$. Не стигат $24499.51 - 18000 = 6499.51$ |

Насоки и подсказки

Ще прочетем входните данни и ще извършим изчисленията, описани в условието на задачата, за да проверим дали ще стигнат парите.

Обработка на входните данни

Нека прочетем внимателно условието и да разгледаме какво се очаква да получим като **входни данни**, какво се очаква да **върнем като резултат**, както и кои са **основните стъпки** при разбиването на логическата схема.

Като за начало, нека обработим и запазим входните данни в **подходящи** за това променливи:

```
decimal budget = decimal.Parse(Console.ReadLine());
string ticketType = Console.ReadLine();
int people = int.Parse(Console.ReadLine());
```

Изчисления

Нека създадем и инициализираме нужните за изчисленията променливи:

```
decimal transportCharges = 0.00M;
decimal moneyForTickets = 0.00M;
decimal moneyDifference = 0.00M;
```

Нека отново прегледаме условието. Трябва да направим **две** различни блок изчисления.

От първите изчисления трябва да разберем каква част от бюджета ще трябва да заделим за **транспорт**. За логиката на тези изчисления забелязваме, че има значение единствено **броят на хората в групата**. Следователно ще направим логическата разбивка спрямо броя на запалянковците.

Ще използваме условна конструкция – поредица от **if-else** блокове.

```
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;
}

```

От вторите изчисления трябва да намерим каква сума ще ни е необходима за закупуване на **билети за групата**. Според условието, това зависи единствено от типа на билетите, които трябва да закупим.

Нека използваме **switch-case** условна конструкция.

```

switch (ticketType)
{
    case "Normal":
        moneyForTickets = people * 249.99M;
        break;
    case "VIP":
        moneyForTickets = people * 499.99M;
        break;
    default:
        moneyForTickets = people * 249.99M;
        break;
}

```

След като сме изчислили какви са транспортните разходи и разходите за билети, ни остава да изчислим крайния резултат и да разберем **ще успее** ли групата от запалянковци да отиде на Евро 2016 или **няма да успее** при така подадените параметри.

За извеждането на резултата, за да си спестим една **else** проверка в конструкцията, приемаме, че групата по подразбиране ще може да отиде на Евро 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)));
}

```

Отпечатване на резултата

Накрая ни остава да покажем изчисления резултат на конзолата.

Тестване в Judge системата

Тествайте решението си: <https://judge.softuni.bg/Contests/Practice/Index/509#3>.

Задача: хотелска стая

Хотел предлага **два** вида стаи: студио и апартамент.

Напишете програма, която изчислява **цената** за целия престой за студио и апартамент. **Цените** зависят от **месеца** на престоя:

| Май и октомври | Юни и септември | Юли и август |
|--------------------------------|-----------------------------------|--------------------------------|
| Студио – 50 лв./нощувка | Студио – 75.20 лв./нощувка | Студио – 76 лв./нощувка |
| Апартамент – 65 лв./нощувка | Апартамент – 68.70 лв./нощувка | Апартамент – 77 лв./нощувка |

Предлагат се и следните **отстъпки**:

- За **студио**, при повече от **7** нощувки през **май и октомври**: **5%** намаление.
- За **студио**, при повече от **14** нощувки през **май и октомври**: **30%** намаление.
- За **студио**, при повече от **14** нощувки през **юни и септември**: **20%** намаление.
- За **апартамент**, при **повече** от **14** нощувки, **без значение** от месеца: **10%** намаление.

Входни данни

Входът се чете от конзолата и съдържа **точно** **два** **реда**:

- На **първия** **ред** е **месецът** – May, June, July, August, September или October.
- На **втория** **ред** е **броят** на **нощувките** – **цяло** **число** в **интервала** [0...200].

Изходни данни

Да се отпечатат на конзолата **два реда**:

- На първия ред: "Apartment: { цена за целият престой } lv".
- На втория ред: "Studio: { цена за целият престой } lv".

Цената да се форматирана с **точност до два символа** след десетичния знак.

Примерен вход и изход

| Вход | Изход | Обяснения |
|-----------|---|---|
| May 15 | Apartment: 877.50 lv. Studio: 525.00 lv. | През май , при повече от 14 нощувки , намаляваме цената на студиото с 30% ($50 - 15 = 35$), а на апартамента – с 10% ($65 - 6.5 = 58.5$). Целият престой в апартамент – 877.50 лв. . Целият престой в студио – 525.00 лв. . |

| Вход | Изход |
|------------|---|
| June 14 | Apartment: 961.80 lv. Studio: 1052.80 lv |

| Вход | Изход |
|--------------|---|
| August 20 | Apartment: 1386.00 lv. Studio: 1520.00 lv. |

Насоки и подсказки

Ще прочетем входните данни и ще извършим изчисленията според описания ценоразпис и правилата за отстъпките и накрая ще отпечатаме резултата.

Обработка на входните данни

Съгласно условието на задачата очакваме да получим два реда входни данни – на първия ред **месец**, през който се планува престой, а на втория – **броя нощувки**.

Нека обработим и запазим входните данни в подходящи за това параметри:

```
string month = Console.ReadLine();
int nights = int.Parse(Console.ReadLine());
```

Изчисления

След това да създадем и инициализираме нужните за изчисленията променливи:

```
decimal studioPrice = 50.00M;
decimal apartmentPrice = 65.00M;
decimal studioRent = 0.00M;
decimal apartmentRent = 0.00M;
```

Разглеждайки отново условието забелязваме, че основната ни логика зависи от това какъв **месец** ни се подава, както и от броя на **нощувките**.

Като цяло има различни подходи и начини да се направят въпросните проверки, но нека се спрем на основна условна конструкция **switch-case**, като в различните **case блокове** ще използваме съответно условни конструкции **if** и **if-else**.

Нека започнем с първата група месеци: **Май** и **Октомври**. За тези два месеца цената на престой е **еднаква** и за двета типа настаняване – в **студио** и в **апартамент**. Съответно остава само да направим вътрешна проверка спрямо **броят нощувки**, за да преизчислим **съответната цена** (ако се налага).

```
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;
}
```

За следващите месеци **логиката и изчисленията** ще са донякъде **идентични**.

```
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;

```

След като изчислихме какви са съответните цени и крайната сума за престоя – нека да си изведем във форматиран вид резултата, като преди това го запишем в изходните ни параметри – **studioInfo** и **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));

```

За изчисленията на изходните параметри използваме метода **decimal.Round(Decimal, Int32)**. Този метод закръгля десетично число до зададен брой цифри след десетичния знак. За целта, подаваме на метода данни от тип **decimal** (**studioRent**, **apartamentPrice**) и цяло число (**int**). В нашия случай ще закръглим десетичното число до две цифри след десетичната точка.

Отпечатване на резултата

Накрая остава да покажем изчислените резултати на конзолата.

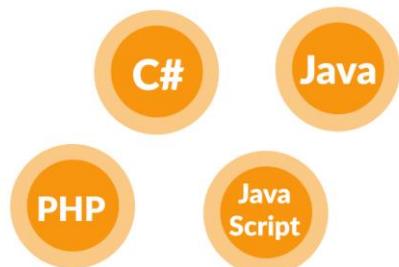
Тестване в Judge системата

Тествайте решението си: <https://judge.softuni.bg/Contests/Practice/Index/509#4>.

Качествено образование,
професия и работа за

Софтуерни инженери

- ✓ Безплатен старт за начинаещи - **присъствено и онлайн**
- ✓ Избор измежду **най-търсените професии** в софтуерната индустрия
- ✓ Съдействие за **кариерен старт**
- ✓ Отлични преподаватели, ментори и студентска общност



Софтуни предоставя съвременно иновативно образование за програмиране, ИТ и дигитални умения на хиляди млади хора. Програмата "**Софтуерен университет**" изгражда истински **професионалисти в света на програмирането**.

Включете се в цялостната програма по софтуерно инженерство на Софтуни, за да усвоите **най-търсените съвременни софтуерни технологии** чрез модерна учебна методика и с много практически задачи и проекти. **Учебният план** е разработен съвместно с ИТ фирмите и изгражда най-търсените от тях умения.

Софтуни работи пряко с компаниите от софтуерната индустрия, съдейтайки на своите студенти за реализацията им като успешни софтуерни инженери.

ПЪТЯТ НА СТУДЕНТА В СОФТУНИ



1 - 1.5 години



1.5 - 2 години



Programming Basics

Кариерен Старт

Дипломиране

70/100 credits

80/100/150 credits

Кандидатствай

softuni.bg/apply

Глава 5.1. Повторения (цикли)

В настоящата глава ще се запознаем с конструкциите за повторение на група команди, известни в програмирането с понятието "цикли". Ще напишем няколко цикъла с използване на оператора **for** в най-простата му форма. Накрая ще решим няколко практически задачи, изискаващи повторение на поредица от действия, като използваме цикли.

Видео

Гледайте видео-урок по тази глава: <https://youtube.com/watch?v=Xjwjk9yS4uw>.

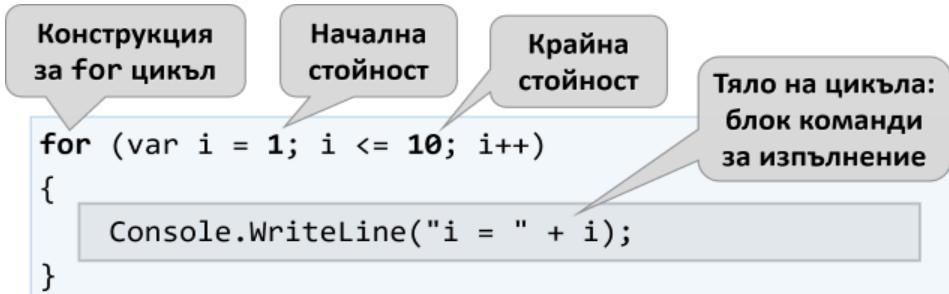
Повторения на блокове код (for цикъл)

В програмирането често пъти се налага да изпълним блок с команди няколко пъти. За целта се използват т.нр. **цикли**. Нека разгледаме един пример за **for** цикъл, който преминава през числата от 1 до 10 и ги отпечатва:

```
for (int i = 1; i <= 10; i++)
{
    Console.WriteLine("i = " + i);
}
```

Цикълът започва с **оператора for** и преминава през всички стойности за дадена променлива в даден интервал, например всички числа от 1 до 10 включително, и за всяка стойност изпълнява поредица от команди.

В декларацията на цикъла може да се зададе **начална стойност** и **краяна стойност**. Тялото на цикъла обикновено се огражда с къдрави скоби **{ }** и представлява блок с една или няколко команди. На фигураната по-долу е показана структурата на един **for** цикъл:



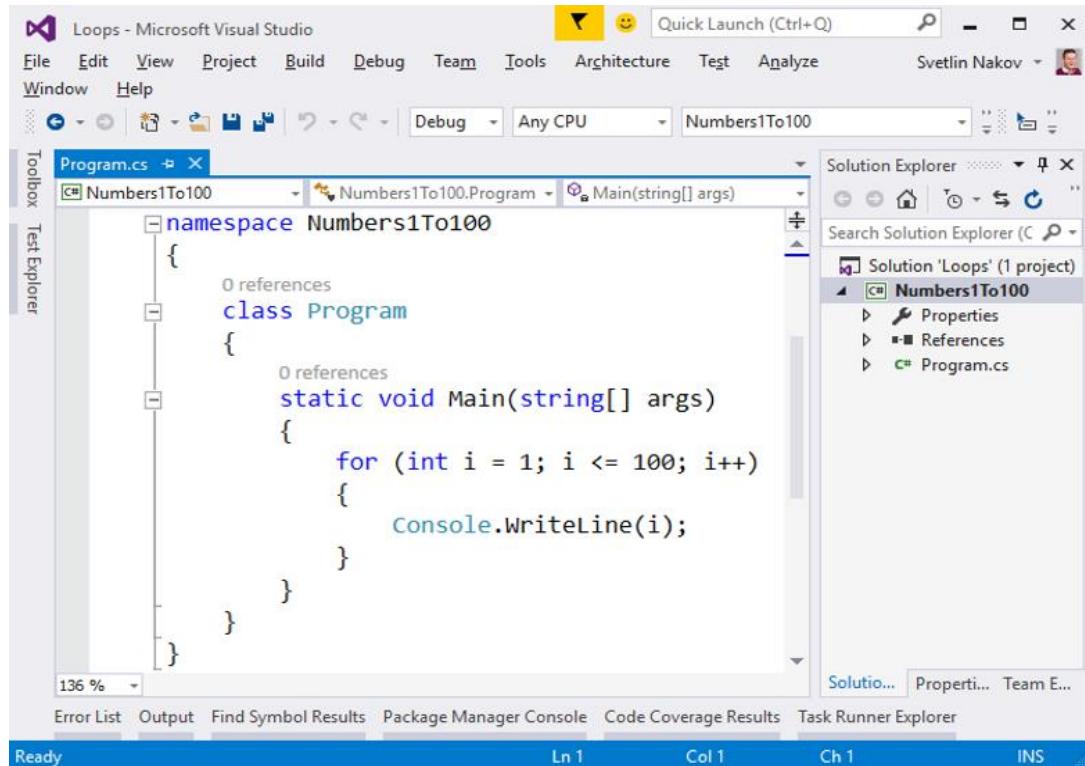
В повечето случаи един **for** цикъл се завърта от **1** до **n** (например от 1 до 10). Целта на цикъла е да се премине последователно през числата 1, 2, 3, ..., **n** и за всяко от тях да се изпълни някакво действие. В примера по-горе променливата **i** приема стойности от 1 до 10 и в тялото на цикъла се отпечатва текущата стойност. Цикълът се повтаря 10 пъти. Всяко от тези повторения се нарича "**итерация**".

Пример: числа от 1 до 100

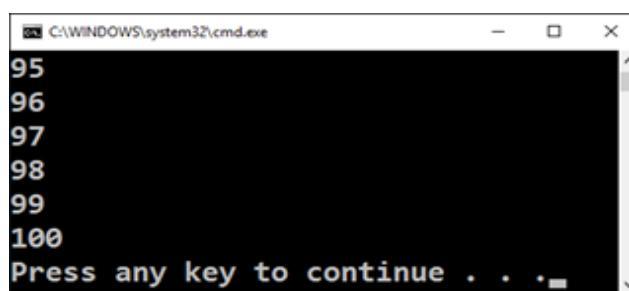
Да се напише програма, която **печатат числата от 1 до 100**. Програмата не приема вход и отпечатва числата от 1 до 100 едно след друго, по едно на ред.

Насоки и подсказки

Можем да решим задачата с **for цикъл**, с който преминаваме с променливата **i** през числата от 1 до 100 и ги печатаме в тялото на цикъла:



Стартираме програмата с [Ctrl+F5] и я тестваме:



Тестване в Judge системата

Тествайте решението си: <https://judge.softuni.bg/Contests/Practice/Index/510#0>. Трябва да получите **100 точки** (напълно коректно решение).

Code Snippet за for цикъл във Visual Studio

Докато програмираме, постоянно се налага да пишем цикли, десетки пъти всеки ден. Затова в повечето среди за разработка (IDE) има шаблони за код (code snippets) за писане на цикли. Един такъв шаблон е **шаблонът за for цикъл във Visual Studio**. Напишете **for** в редактора за C# код във Visual Studio и натиснете два пъти [Tab]. Visual Studio ще разгъне за вас шаблон и ще напише цялостен **for** цикъл:



Опитайте сами, за да усвоите умението да ползвате шаблона за код за **for** цикъл във Visual Studio.

Пример: числа до 1000, завършващи на 7

Да се напише програма, която намира всички числа в интервала [1...1000], които завършват на 7.

Насоки и подсказки

Задачата можем да решим като комбинираме **for** цикъл за преминаваме през числата от 1 до 1000 и проверка за всяко число дали завършва на 7. Има и други решения, разбира се, но нека решим задачата чрез завъртане на цикъл + проверка:

```
for (int i = 1; i <= 1000; i++)
{
    if (i % 10 == 7)
    {
        // TODO: print i
    }
}
```

Тестване в Judge системата

Тествайте решението си: <https://judge.softuni.bg/Contests/Practice/Index/510#1>.

Пример: всички латински букви

Да се напише програма, която отпечатва буквите от латинската азбука: **a, b, ..., z**.

Насоки и подсказки

Полезно е да се знае, че **for** циклите не работят само с числа. Може да решим задачата като завъртим **for** цикъл, който преминава последователно през всички букви от латинската азбука:

```
Console.WriteLine("Latin alphabet:");
for (var letter = 'a'; letter <= 'z'; letter++)
{
    Console.WriteLine(" " + letter);
}
Console.WriteLine();
```

Тестване в Judge системата

Тествайте решението си: <https://judge.softuni.bg/Contests/Practice/Index/510#2>.

Пример: сумиране на числа

Да се напише програма, която **въвежда n** цели числа и ги сумира.

- От първия ред на входа се въвежда броят числа **n**.
- От следващите **n** реда се въвежда по едно число.
- Числата се сумират и накрая се отпечатва резултатът.

Примерен вход и изход

| Вход | Изход |
|------|-------|
| 2 | |
| 10 | |
| 20 | 30 |

| Вход | Изход |
|------|-------|
| 3 | |
| -10 | |
| -20 | |
| -30 | -60 |

| Вход | Изход |
|------|-------|
| 4 | |
| 45 | |
| -20 | |
| 7 | |
| 11 | 43 |

| Вход | Изход |
|----------|-------|
| 1 999 | 999 |

| Вход | Изход |
|------|-------|
| 0 | 0 |

Насоки и подсказки

Можем да решим задачата за сумиране на числа по следния начин:

- Четем входното число **n**.
- Започваме първоначално със сума **sum = 0**.
- Въртим цикъл от 1 до **n**. На всяка стъпка от цикъла четем число **num** и го добавяме към сумата **sum**.
- Накрая отпечатваме получената сума **sum**.

Ето и сорс кода на решението:

```

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);

```

Тестване в Judge системата

Тествайте решението си: <https://judge.softuni.bg/Contests/Practice/Index/510#3>.

Пример: най-голямо число

Да се напише програма, която въвежда **n** цели числа ($n > 0$) и намира **най-голямото** измежду тях. На първия ред на входа се въвежда броят числа **n**. След това се въвеждат самите числа, по едно на ред.

Примерен вход и изход

| Вход | Изход |
|------|-------|
| 2 | |
| 100 | 100 |
| 99 | |

| Вход | Изход |
|------|-------|
| 3 | |
| -10 | |
| 20 | 20 |
| -30 | |

| Вход | Изход |
|------|-------|
| 4 | |
| 45 | |
| -20 | |
| 7 | 99 |
| 99 | |

| Вход | Изход |
|------|-------|
| 1 | |
| 999 | 999 |

| Вход | Изход |
|------|-------|
| 2 | |
| -1 | |
| -2 | -1 |

Насоки и подсказки

Първо въвеждаме едно число **n** (броят числа, които предстои да бъдат въведени). Задаваме на текущия максимум **max** първоначална неутрална стойност, например **-1000000000000000** (или **int.MinValue**). С помощта на **for цикъл**, чрез който итерираме **n-1** пъти, прочитаме по едно цяло число **num**. Ако прочетеното число **num**

е по-голямо от текущия максимум **max**, присвояваме стойността на **num** в променливата **max**. Накрая, в **max** трябва да се е запазило най-голямото число. Отпечатваме го на конзолата.

```
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);
```

Тестване в Judge системата

Тествайте решението си: <https://judge.softuni.bg/Contests/Practice/Index/510#4>.

Пример: най-малко число

Да се напише програма, която въвежда **n** цели числа ($n > 0$) и намира най-малкото измежду тях. Въвеждат се броят числа **n** и след тях още **n** числа по едно на ред.

Примерен вход и изход

| Вход | Изход | Вход | Изход | Вход | Изход |
|----------------|-------|-----------------------|-------|---------------------------|-------|
| 2 100 99 | 99 | 3 -10 20 -30 | -30 | 4 45 -20 7 99 | -20 |

Насоки и подсказки

Задачата е абсолютно аналогична с предходната, само че започване с друга неутрална начална стойност.

```
Console.WriteLine("n = ");
var n = int.Parse(Console.ReadLine());
```

```

var min = 10000000000000000;

for (int i = 1; i <= n; i++)
{
    var num = int.Parse(Console.ReadLine());
    if (num < min)
    {
        min = num;
    }
}

Console.WriteLine("min = " + min);

```

Тестване в Judge системата

Тествайте решението си: <https://judge.softuni.bg/Contests/Practice/Index/510#5>.

Пример: лява и дясна сума

Да се напише програма, която въвежда $2 * n$ цели числа и проверява дали сумата на първите n числа (лява сума) е равна на сумата на вторите n числа (дясна сума).

При равенство да се печата "Yes" + сумата, иначе да се печата "No" + разликата. Разликата се изчислява като положително число (по абсолютна стойност). Форматът на изхода трябва да е като в примерите по-долу.

Примерен вход и изход

| Вход | Изход | Вход | Изход |
|------|----------------|------|--------------|
| 2 | | 2 | |
| 10 | | 90 | |
| 90 | Yes, sum = 100 | 9 | No, diff = 1 |
| 60 | | 50 | |
| 40 | | 50 | |

Насоки и подсказки

Първо въвеждаме числото n , след това първите n числа (лявата половина) и ги сумираме. Продължаваме с въвеждането на още n числа (дясната половина) и наричаме и тяхната сума. Изчисляваме разликата между намерените суми по абсолютна стойност: `Math.Abs(leftSum - rightSum)`. Ако разликата е 0, отпечатваме "Yes" + сумата, в противен случай – отпечатваме "No" + разликата.

```

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);
}

```

Тестване в Judge системата

Тествайте решението си: <https://judge.softuni.bg/Contests/Practice/Index/510#6>.

Пример: четна / нечетна сума

Да се напише програма, която въвежда **n** цели числа и проверява дали **сумата на числата на четни позиции** е равна на **сумата на числата на нечетни позиции**. При равенство печата "Yes" + **сумата**, иначе печата "No" + **разликата**. Разликата се изчислява по абсолютна стойност. Форматът на изхода трябва да е като в примерите по-долу.

Примерен вход и изход

| Вход | Изход | Вход | Изход | Вход | Изход |
|------|----------|------|----------|------|----------|
| 4 | | 4 | | 3 | |
| 10 | | 3 | | 5 | |
| 50 | Yes | 5 | No | 8 | |
| 60 | Sum = 70 | 1 | Diff = 1 | 1 | Diff = 2 |
| 20 | | -2 | | | |

Насоки и подсказки

Въвеждаме числата едно по едно и изчисляваме двете **суми** (на числата на **четни** позиции и на числата на **нечетни** позиции). Както в предходната задача, изчисляваме абсолютната стойност на разликата и отпечатваме резултата ("Yes" + сумата при разлика 0 или "No" + разликата в противен случай).

```
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
```

Тестване в Judge системата

Тествайте решението си: <https://judge.softuni.bg/Contests/Practice/Index/510#7>.

Пример: сумиране на гласните букви

Да се напише програма, която въвежда **текст** (стринг), изчислява и отпечатва **сумата от стойностите на гласните букви** според таблицата вдясно.

| | | | | |
|---|---|---|---|---|
| a | e | i | o | u |
| 1 | 2 | 3 | 4 | 5 |

Примерен вход и изход

| Вход | Изход |
|-------|----------------------|
| hello | 6 (e+o = 2+4 = 6) |

| Вход | Изход |
|--------|--------------------------|
| bamboo | 9 (a+o+o = 1+4+4 = 9) |

| Вход | Изход | Вход | Изход |
|------|--------------|------|----------------------|
| hi | 3 (i = 3) | beer | 4 (e+e = 2+2 = 4) |

Насоки и подсказки

Прочитаме входния текст **s**, зануляваме сумата и завъртаме цикъл от 0 до **s.Length - 1** (дължината на текста -1). Проверяваме всяка буква **s[i]** дали е гласна и съответно добавяме към сумата стойността ѝ.

```

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);

```

Тестване в Judge системата

Тествайте решението си: <https://judge.softuni.bg/Contests/Practice/Index/510#8>.

Какво научихме от тази глава?

Можем да повтаряме блок код с **for** цикъл:

```
for (int i = 0; i <= 10; i++)
{
    Console.WriteLine("i = " + i);
}
```

Можем да четем поредица от **n** числа от конзолата:

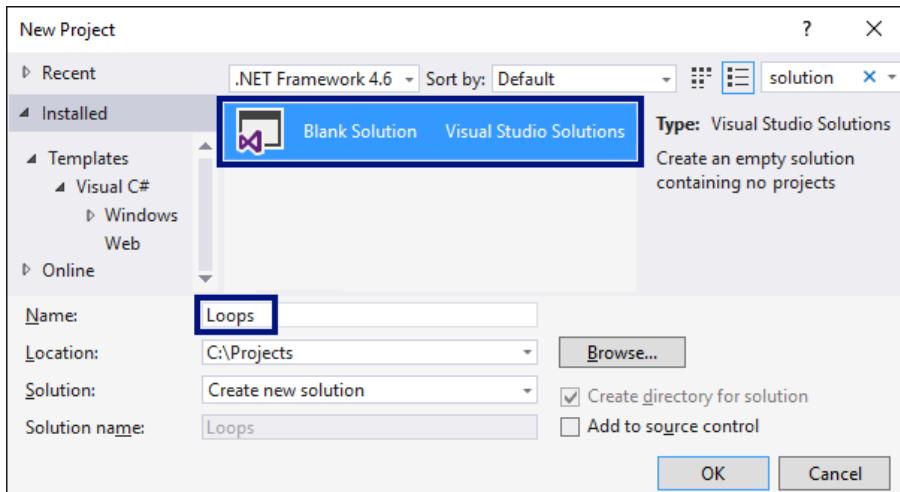
```
var n = int.Parse(Console.ReadLine());
for (int i = 0; i < n; i++)
{
    var num = int.Parse(Console.ReadLine());
}
```

Упражнения: повторения (цикли)

След като се запознахме с циклите, идва време да затвърдим знанията си на практика, а както знаете, това става с много писане на код. Да решим няколко задачи за упражнение.

Празно Visual Studio решение (Blank Solution)

Създаваме празно решение (**Blank Solution**) във Visual Studio, за да организираме по-добре задачите за упражнение. Целта на този **blank solution** е да съдържа **по един проект за всяка задача** от упражненията.



Задаваме да се стартира по подразбиране текущият проект (не първият в решението). Кликваме с десен бутон на мишката върху Solution 'Loops' -> [Set StartUp Projects...] -> [Current selection].

Задача: елемент, равен на сумата на останалите

Да се напише програма, която въвежда **n** цели числа и проверява дали сред тях съществува число, равно на сумата на всички останали. Ако има такъв елемент, се отпечатва "Yes" + неговата стойност, в противен случай – "No" + разликата между най-големия елемент и сумата на останалите (по абсолютна стойност).

Примерен вход и изход

| Вход | Изход | Коментар | Вход | Изход | Коментар |
|---------------------------------------|-----------------|------------------------------|-------------------|----------------|----------------------|
| 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$ |

| Вход | Изход | Вход | Изход | Вход | Изход |
|------------------|----------------|------------------|----------------|-----------------------|----------------|
| 3 1 1 1 | No Diff = 1 | 3 5 5 1 | No Diff = 1 | 4 6 1 2 3 | Yes Sum = 6 |

Насоки и подсказки

Трябва да изчислим сумата на всички елементи, да намерим **най-големия** от тях и да проверим търсеното условие.

Тестване в Judge системата

Тествайте решението си: <https://judge.softuni.bg/Contests/Practice/Index/510#9>.

Задача: четни / нечетни позиции

Напишете програма, която чете **n** числа и пресмята **сумата, минимума и максимума** на числата на **четни и нечетни** позиции (броим от 1). Когато няма минимален или максимален елемент, отпечатайте "No".

Примерен вход и изход

| Вход | Изход |
|------|--|
| 6 | |
| 2 | |
| 3 | |
| 5 | |
| 4 | |
| 2 | |
| 1 | |
| | OddSum=9, OddMin=2, OddMax=5, EvenSum=8, EvenMin=1, EvenMax=4 |

| Вход | Изход |
|------|---|
| 2 | |
| 1.5 | |
| -2.5 | |
| | OddSum=1.5, OddMin=1.5, OddMax=1.5, EvenSum=-2.5, EvenMin=-2.5, EvenMax=-2.5 |

| Вход | Изход |
|------|--|
| 1 | |
| 1 | |
| | OddSum=1, OddMin=1, OddMax=1, EvenSum=0, EvenMin>No, EvenMax>No |

| Вход | Изход |
|------|--|
| 3 | |
| -1 | |
| -2 | |
| -3 | |
| | OddSum=-4, OddMin=-3, OddMax=-1, EvenSum=-2, EvenMin=-2, EvenMax=-2 |

| Вход | Изход |
|------|---|
| 1 | |
| -5 | |
| | OddSum=-5, OddMin=-5, OddMax=-5, EvenSum=0, EvenMin>No, EvenMax>No |

| Вход | Изход |
|------|---|
| 5 | |
| 3 | |
| -2 | |
| 8 | |
| 11 | |
| -3 | |
| | OddSum=8, OddMin=-3, OddMax=8, EvenSum=9, EvenMin=-2, EvenMax=11 |

Насоки и подсказки

Задачата обединява няколко предходни задачи: намиране на **минимум**, **максимум** и **сума**, както и обработка на елементите от **четни и нечетни позиции**. Припомнете си ги.

В тази задача е по-добре да се работи с **дробни числа** (не цели). Сумата, минимумът и максимумът също са дробни числа. Трябва да използваме **неутрална начална стойност** при намиране на минимум / максимум, например **1000000000.0** и **-1000000000.0**. Ако получимнакрая неутралната стойност, печатаме "**No**".

Тестване в Judge системата

Тествайте решението си: <https://judge.softuni.bg/Contests/Practice/Index/510#10>.

Задача: еднакви двойки

Дадени са **$2 * n$ числа**. Първото и второто формират **двойка**, третото и четвъртото също и т.н. Всяка двойка има **стойност** – сумата от съставящите я числа. Напишете програма, която проверява **дали всички двойки имат еднаква стойност**.

В случай, че е еднаква отпечатайте "**Yes, value=...**" + **стойността**, в противен случай отпечатайте **максималната разлика** между две последователни двойки в следния формат – "**No, maxdiff=...**" + **максималната разлика**.

Входът се състои от число **n**, следвано от **$2 * n$ цели числа**, всички по едно на ред.

Примерен вход и изход

| Вход | Изход | Коментар | Вход | Изход | Коментар |
|----------------------------------|--------------|--|-----------------------|---------------|--|
| 3 1 2 0 3 4 -1 | Yes, value=3 | стойности = {3, 3, 3} еднакви стойности | 2 1 2 2 2 | No, maxdiff=1 | стойности = {3, 4} разлики = {1} макс. разлика = 1 |

| Вход | Изход | Коментар | Вход | Изход | Коментар |
|-------------------------|---------------|---|-------------|---------------|---|
| 2 -1 2 0 -1 | No, maxdiff=2 | стойности = {1, -1} разлики = {2} макс. разлика = 2 | 1 5 5 | Yes, value=10 | стойности = {10} една стойност еднакви стойности |

| Вход | Изход | Вход | Изход |
|-------------------------|---------------|---|---------------|
| 2 -1 0 0 -1 | Yes, value=-1 | 4 1 1 3 1 2 2 0 0 | No, maxdiff=4 |

Насоки и подсказки

Прочитаме входните числа **по двойки**. За всяка двойка пресмятаме **сумата** ѝ. Докато четем входните двойки, за всяка двойка, без първата, трябва да пресметнем **разликата с предходната**. За целта е необходимо да пазим в отделна променлива сумата на предходната двойка. Накрая намираме **най-голямата разлика** между две двойки. Ако е 0, печатаме "Yes" + стойността, в противен случай – "No" + разликата.

Тестване в Judge системата

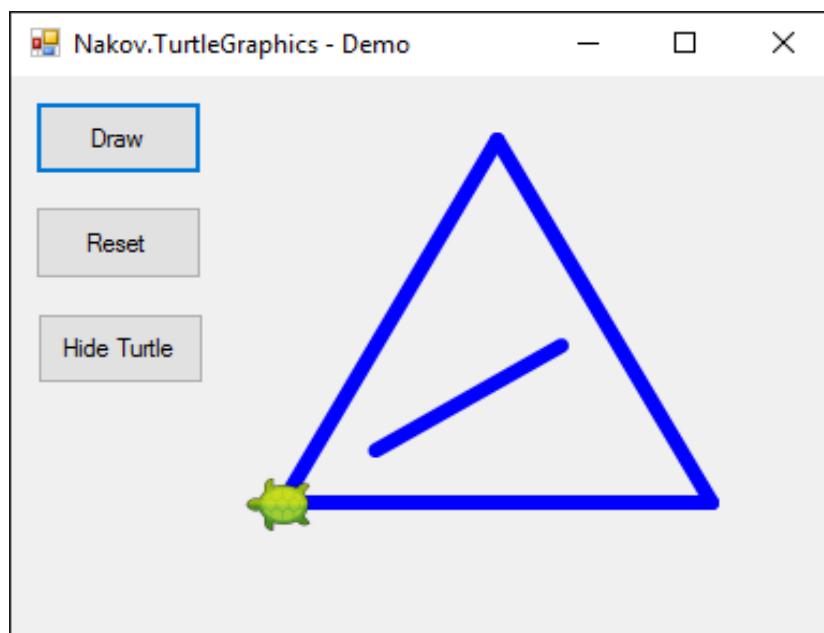
Тествайте решението си: <https://judge.softuni.bg/Contests/Practice/Index/510#11>.

Упражнения: графични и уеб приложения

В настоящата глава се запознахме с **циките** като конструкция в програмирането, която ни позволява да повтаряме многократно дадено действие или група от действия. Сега нека си поиграем с тях. За целта ще начертаем няколко графични фигури, които се състоят от голем брой повтарящи се графични елементи, но този път не на конзолата, а в графична среда, използвайки "графика с костенурка". Ще е интересно. И никак не е сложно. Опитайте!

Задача: чертане с костенурка – графично GUI приложение

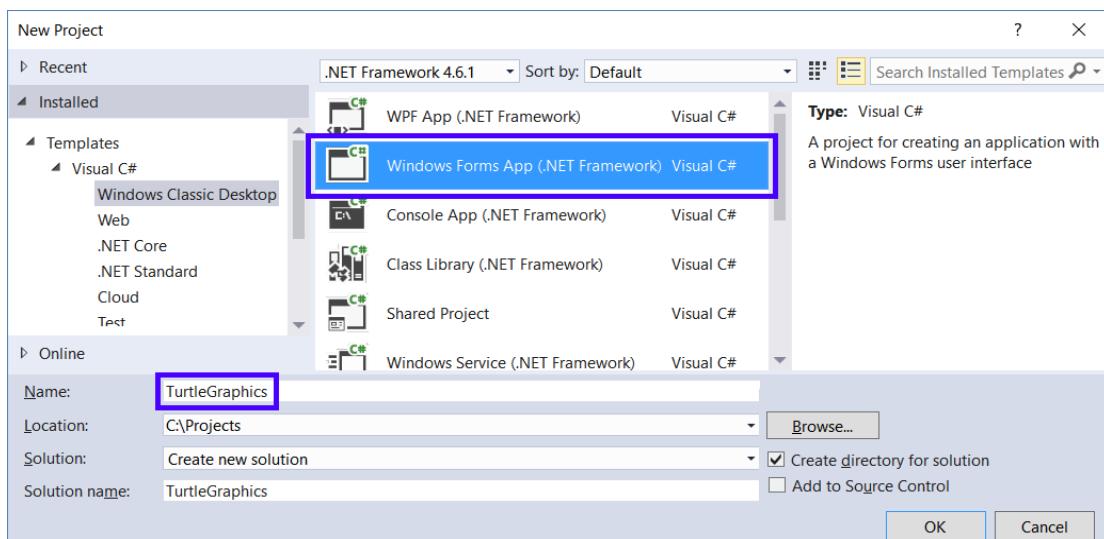
Целта на следващото упражнение е да си поиграем с една **библиотека за рисуване**, известна като "графика с костенурка" (turtle graphics). Ще изградим графично приложение, в което ще **рисуваме различни фигури**, придвижвайки нашата "костенурка" по екрана чрез операции от типа "отиди напред 100 позиции", "завърти се надясно на 30 градуса", "отиди напред още 50 позиции". Приложението ще изглежда приблизително така:



Нека първо се запознаем с **концепцията за рисуване "Turtle Graphics"**. Може да разгледаме следните източници:

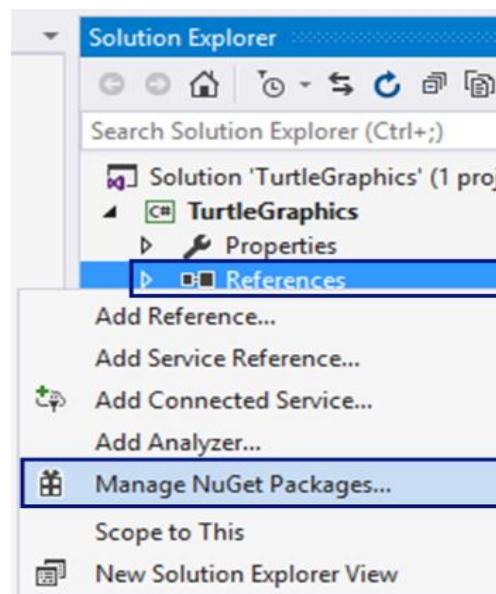
- Дефиниция на понятието "turtle graphics":
<http://c2.com/cgi/wiki?TurtleGraphics>
- Статия за "turtle graphics" в Wikipedia:
https://en.wikipedia.org/wiki/Turtle_graphics
- Интерактивен онлайн инструмент за чертане с костенурка:
<https://blockly-games.appspot.com/turtle>

Започваме, като създаваме нов C# Windows Forms проект:

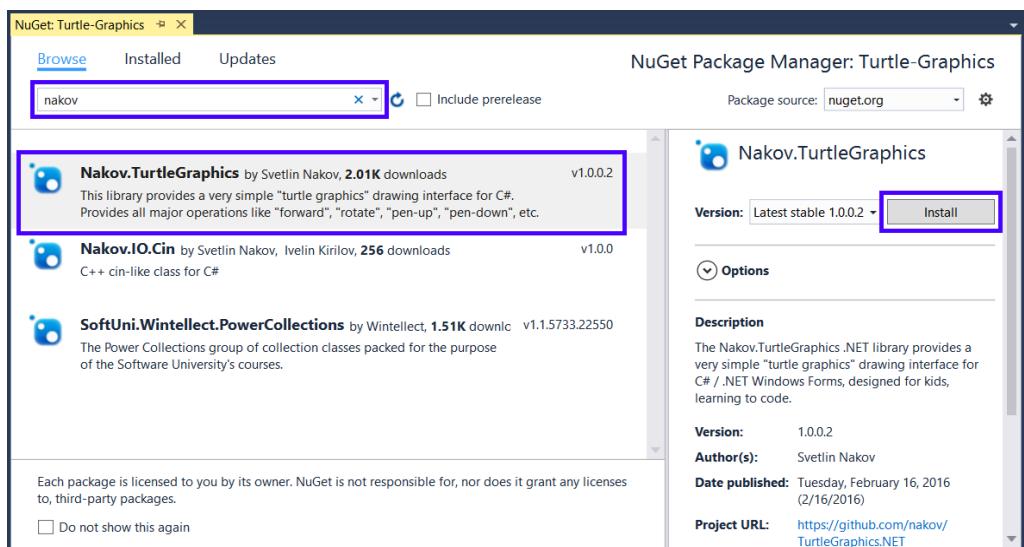


Инсталираме NuGet пакета "Nakov.TurtleGraphics" към нашия нов Windows Forms проект. От Visual Studio може да се добавят **външни библиотеки** (пакети) към съществуващ C# проект. Те добавят допълнителна функционалност към нашите приложения. Официалното хранилище (repository) за C# библиотеки се поддържа от Microsoft и се нарича [NuGet](http://www.nuget.org) (<http://www.nuget.org>).

Кликаме с десен бутон на мишката върху проекта в Solution Explorer и избираме [Manage NuGet Packages...]:



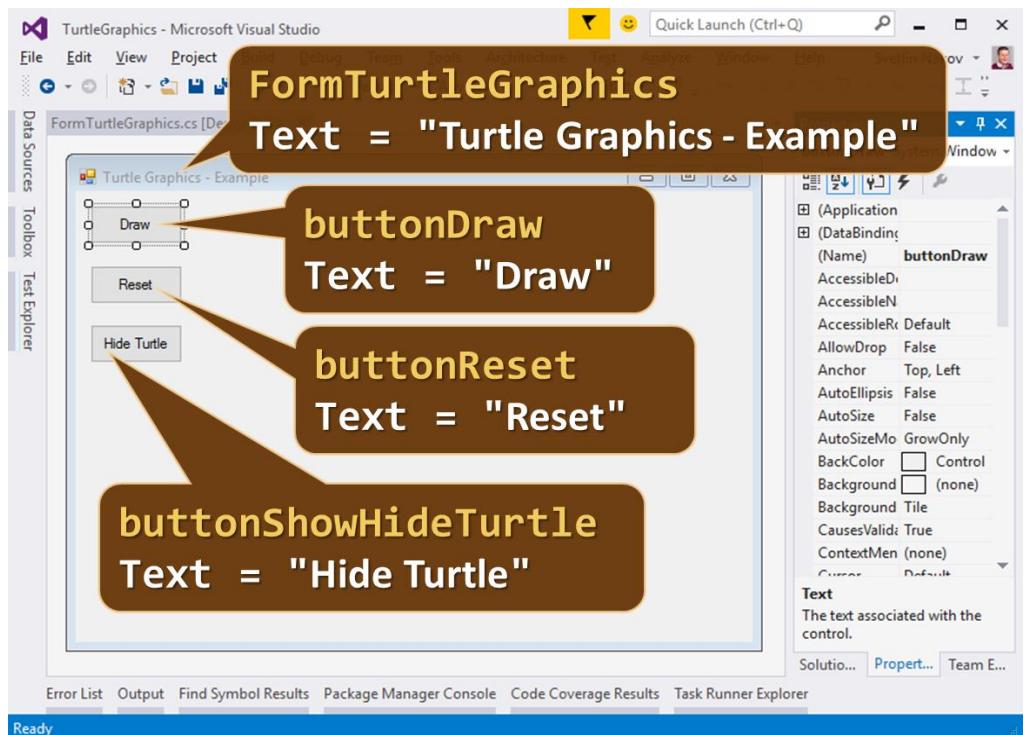
Ще се отвори прозорец за търсене и инсталација на NuGet пакети. Нека потърсим пакети по ключова дума **nakov**. Ще излязат няколко пакета. От тях избираме пакет **Nakov.TurtleGraphics**. Натискаме [Install], за да го инсталуваме към нашия C# проект:



Към нашия C# проект вече е включена външната библиотека **Nakov.TurtleGraphics**. Тя дефинира клас **Turtle**, който представлява костенурка за рисуване. За да го използваме, трябва да добавим в C# кода за нашата форма (**Form1.cs**). Добавяме следния код, най-отгоре в началото на файла:

```
using Nakov.TurtleGraphics;
```

Сега трябва да сложим **три бутона** във формата и да променим **имената и свойствата** им, както е посочено по-долу:

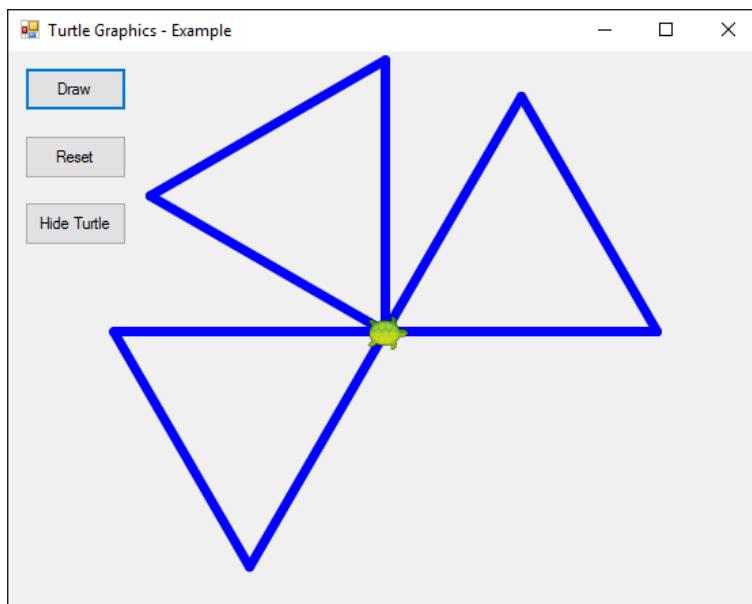


Кликваме два пъти върху бутона [Draw], за да въведем кода, който да се изпълни при натискането му. Пишем следния код:

```
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);
}
```

Този код мести и върти костенурката, която в началото е в центъра на екрана (в средата на формата), и чертае равностранен триъгълник. Може да го редактирате и да си поиграете с него.

Стартираме приложението с [Ctrl+F5] и го пробваме дали работи (натискаме [Draw] бутона няколко пъти):



Сега може да променим и усложним кода на костенурката:

```
// Assign a delay to visualize the drawing process
Turtle.Delay = 200;

// Draw a equilateral triangle
Turtle.Rotate(30);
Turtle.Forward(200);
Turtle.Rotate(120);
```

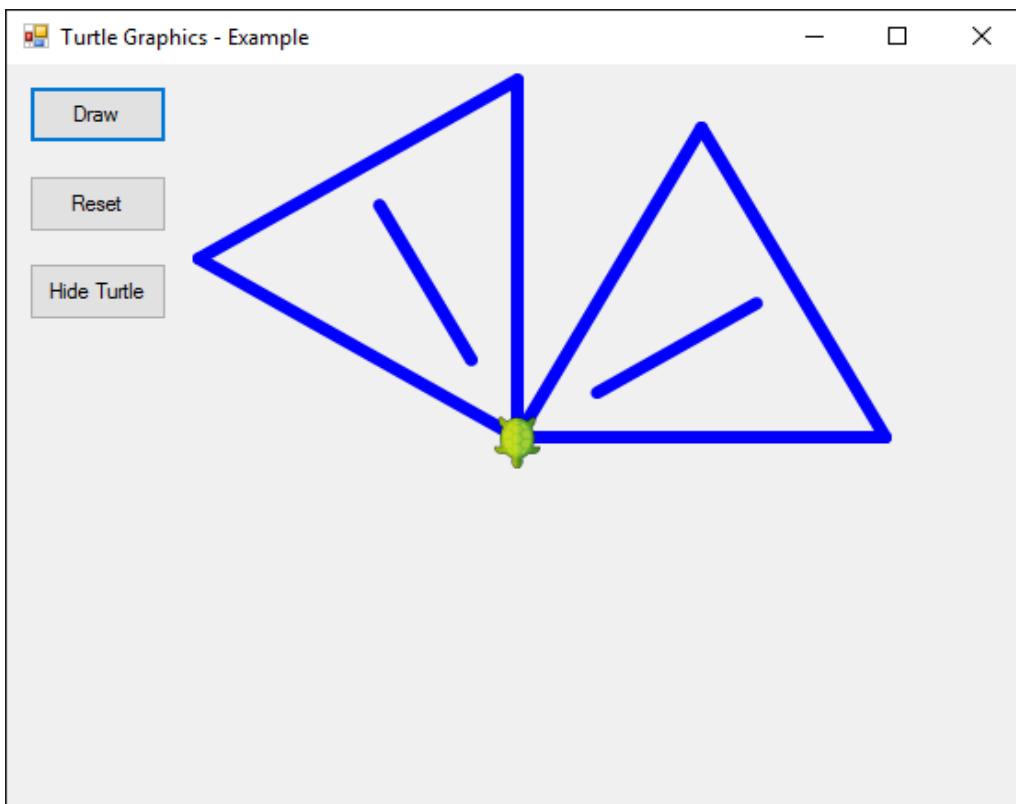
```

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);

```

Отново **стартираме** приложението с [Ctrl+F5]. Тестваме дали работи новата програма за костенурката:



Вече нашата костенурката чертае по-сложни фигури чрез приятно анимирано движение.

Нека напишем кода и за останалите два бутона. Целта на бутона [Reset] е да изтрие графиката и да започне да чертае на чисто:

```
private void buttonReset_Click(object sender, EventArgs e)
{
    Turtle.Reset();
}
```

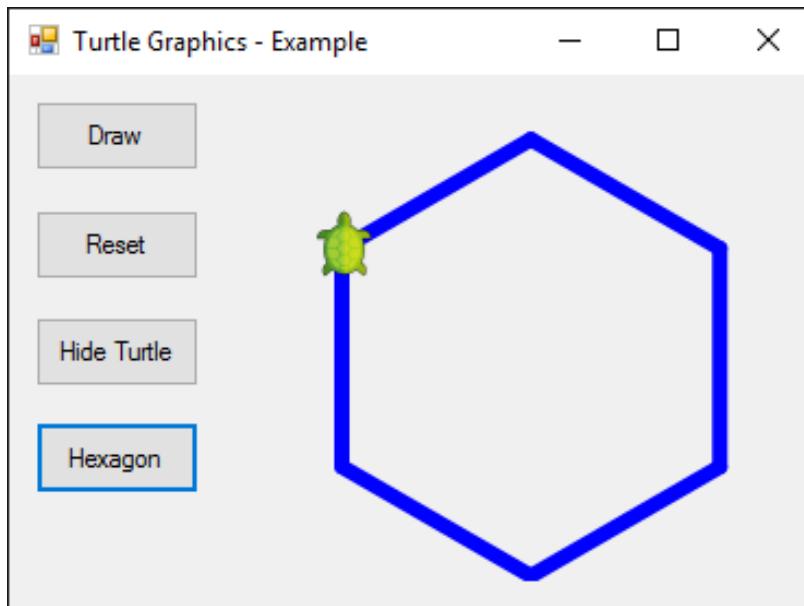
Целта на бутона [Show / Hide Turtle] е да показва или скрива костенурката:

```
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";
    }
}
```

Стартираме пак приложението с [Ctrl+F5] и го тестваме дали работи коректно.

Задача: * чертане на шестоъгълник с костенурката

Добавете бутон [Hexagon], който чертае правилен шестоъгълник:



Подсказки:

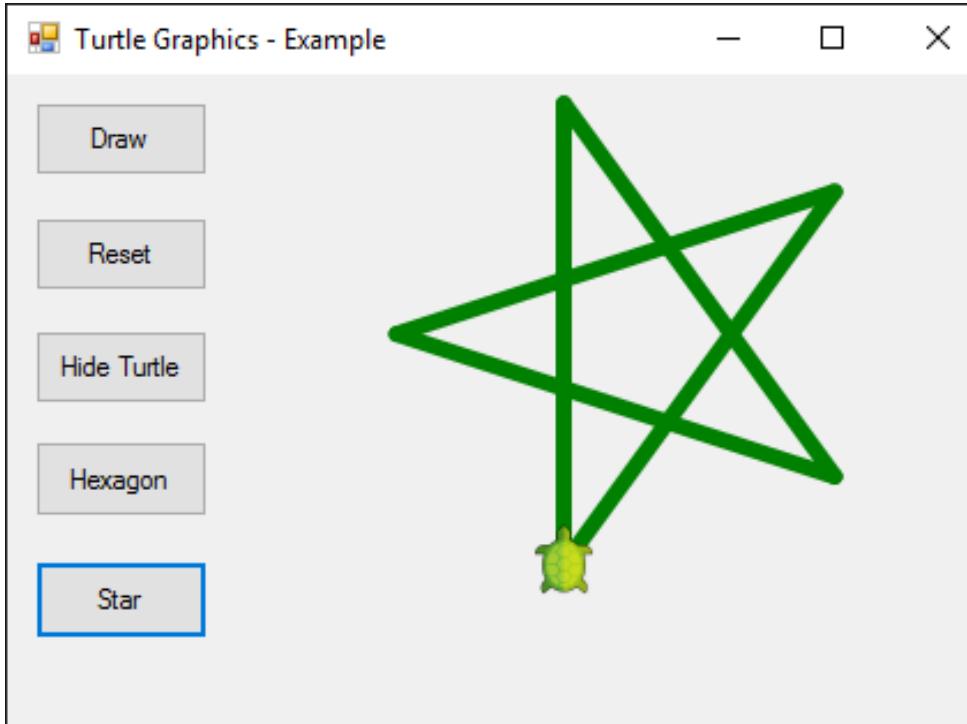
Сложете нов бутона във формата, задайте му съмислено име, сменете му заглавието и кликнете два пъти с мишката, за да напишете кода, който ще се изпълни при натискане на бутона.

За изчертаване на шестоъгълник в цикъл повторете 6 пъти следното:

- Ротация на 60 градуса.
- Движение напред 100.

Задача: * чертане на звезда с костенурката

Добавете бутона [Star], който чертае звезда с 5 върха (петолъчка), като на фигурата по-долу:



Подсказка:

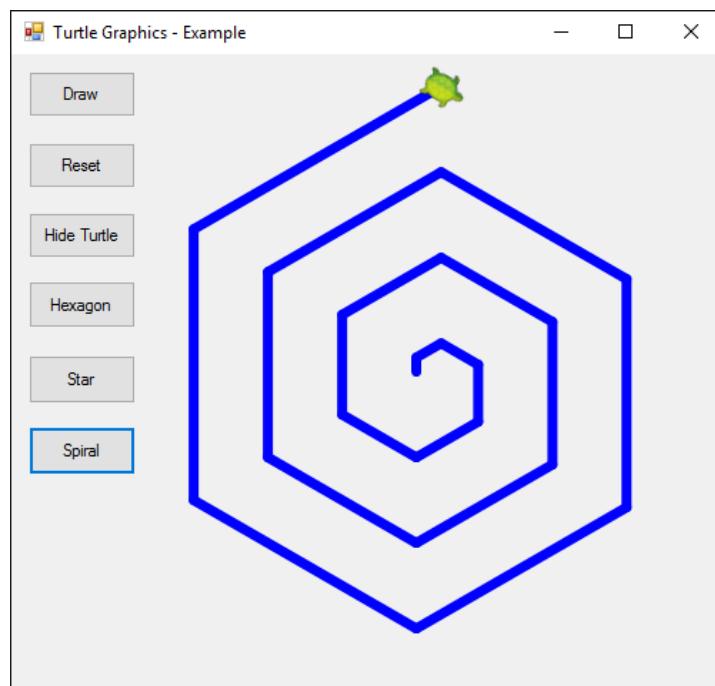
Сменете цвета: **Turtle.PenColor = Color.Green**.

В цикъл повторете 5 пъти следното:

- Движение напред 200.
- Ротация на 144 градуса.

Задача * чертане на спирала с костенурката

Добавете бутона [Spiral], който чертае спирала с 20 върха като на фигурата:

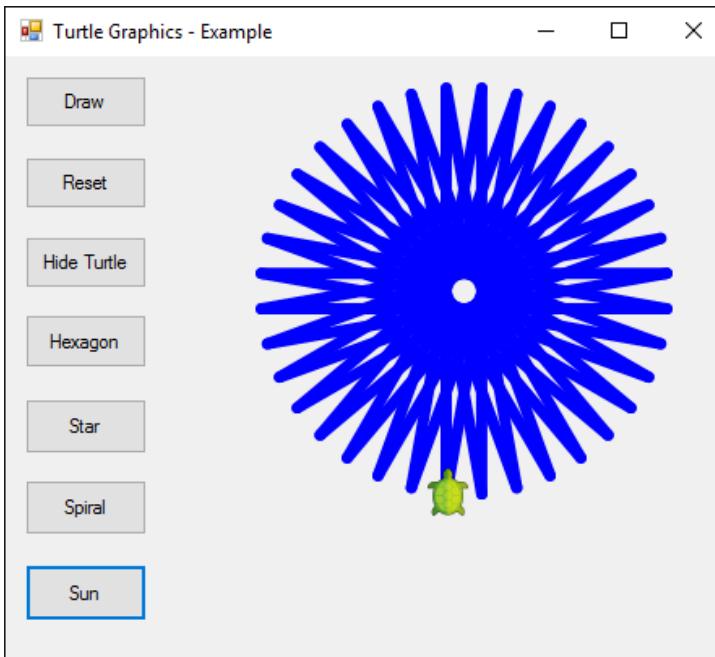


Подсказка:

Чертайте в цикъл, като движите напред и завъртате. С всяка стъпка увеличавайте постепенно дължината на движението напред и завъртайте на 60 градуса.

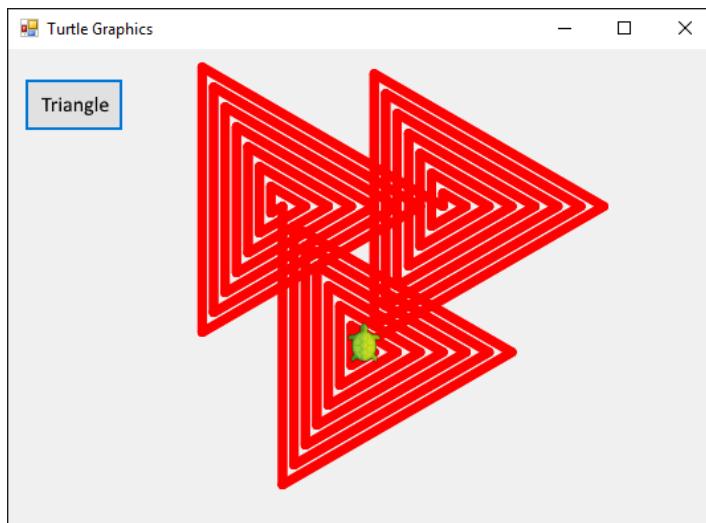
Задача: * чертане на слънце с костенурката

Добавете бутон [Sun], който чертае слънце с 36 върха като на фигурата по-долу:



Задача: * чертане на спирален триъгълник с костенурката

Добавете бутона [Triangle], който чертае три триъгълника с по 22 върха като на фигурата по-долу:



Подсказка:

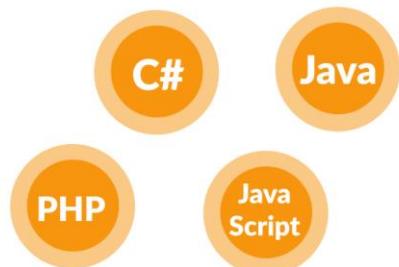
Чертайте в цикъл като движите напред и завъртате. С всяка стъпка увеличавайте с 10 дължината на движението напред и завъртайте на 120 градуса. Повторете 3 пъти за трите триъгълника.

Ако имате проблеми с примерния проект по-горе, **гледайте видеото** в началото на тази глава. Там приложението е направено на живо стъпка по стъпка с много обяснения. Или питайте във **форума на СофтУни**: <https://softuni.bg/forum>.

Качествено образование,
професия и работа за

Софтуерни инженери

- ✓ Безплатен старт за начинаещи - **присъствено и онлайн**
- ✓ Избор измежду **най-търсените професии** в софтуерната индустрия
- ✓ Съдействие за **кариерен старт**
- ✓ Отлични преподаватели, ментори и студентска общност



Софтуни предоставя съвременно иновативно образование за програмиране, ИТ и дигитални умения на хиляди млади хора. Програмата "**Софтуерен университет**" изгражда истински **професионалисти в света на програмирането**.

Включете се в цялостната програма по софтуерно инженерство на Софтуни, за да усвоите **най-търсените съвременни софтуерни технологии** чрез модерна учебна методика и с много практически задачи и проекти. **Учебният план** е разработен съвместно с ИТ фирмите и изгражда най-търсените от тях умения.

Софтуни работи пряко с компаниите от софтуерната индустрия, съдейтайки на своите студенти за реализацията им като успешни софтуерни инженери.

ПЪТЯТ НА СТУДЕНТА В СОФТУНИ



1 - 1.5 години



1.5 - 2 години



Programming Basics

Кариерен Старт

Дипломиране

70/100 credits

80/100/150 credits

Кандидатствай

softuni.bg/apply

Глава 5.2. Повторения (цикли) – изпитни задачи

В предходната глава научихме как да изпълним даден блок от команди **повече от веднъж**. Затова въведохме **for цикъл** и разглеждахме някои от основните му приложения. Целта на настоящата глава е да затвърдим знанията си, решавайки няколко по-сложни задачи с цикли, давани на приемни изпити. За някои от тях ще покажем примерни подробни решения, а за други ще оставим само напътствия. Преди да се захванем за работа е добре да си припомним конструкцията на **цикъла for**:



For циклите се състоят от:

- Инициализационен блок, в който се декларира променливата-брояч (**var i**) и се задава нейна начална стойност.
- Условие за повторение (**i <= 10**), изпълняващо се веднъж, преди всяка итерация на цикъла.
- Обновяване на брояча (**i++**) – този код се изпълнява след всяка итерация.
- Тяло на цикъла – съдържа произволен блок със сорс код.

Изпитни задачи

Да решим няколко задачи с цикли от изпити в СофтУни.

Задача: хистограма

Дадени са **n** цели числа в интервала [1...1000]. От тях някакъв процент **p1** са под 200, процент **p2** са от 200 до 399, процент **p3** са от 400 до 599, процент **p4** са от 600 до 799 и останалите **p5** процента са от 800 нагоре. Да се напише програма, която изчислява и отпечатва процентите **p1**, **p2**, **p3**, **p4** и **p5**.

Пример: имаме **n = 20** числа: 53, 7, 56, 180, 450, 920, 12, 7, 150, 250, 680, 2, 600, 200, 800, 799, 199, 46, 128, 65. Получаваме следното разпределение и визуализация:

| Група | Числа | Брой числа | Процент |
|------------|---|------------|--------------------------------|
| < 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 | $p5 = 2 / 20 * 100 = 10.00\%$ |

Входни данни

На първия ред от входа стои цялото число **n** ($1 \leq n \leq 1000$), което представлява броя редове с числа, които ще ни бъдат подадени. На следващите **n реда** стои по едно цяло число в интервала [1...1000] – числата, върху които да бъде изчислена хистограмата.

Изходни данни

Да се отпечата на конзолата **хистограма от 5 реда**, всеки от които съдържа число между 0% и 100%, форматирано с точност две цифри след десетичния знак (например 25.00%, 66.67%, 57.14%).

Примерен вход и изход

| Вход | Изход | Вход | Изход | Вход | Изход |
|------|--------|------|--------|------|--------|
| 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% |

| Вход | Изход | Вход | Изход |
|------|--------|------|--------|
| 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 | |

Насоки и подсказки

Програмата, която решава този проблем, можем да разделим мислено на 3 части:

- **Прочитане на входните данни** – това включва прочитане на числото **n**, последвано от **n** на брой цели числа, всяко на отделен ред.
- **Обработка на входните данни** – в случая това означава разпределяне на числата по групи и изчисляване на процентното разделение по групи.
- **Извеждане на краен резултат** – отпечатване на хистограмата на конзолата в посочения формат.

Преди да продължим напред ще направим едно малко отклонение от настоящата тема, а именно ще споменем накратко, че в програмирането всяка променлива е от някакъв **тип данни**. В тази задача ще използваме числовите типове **int** за **цели числа** и **double** за **дробни**. Често, за улеснение, програмистите изпускат изричното уточняване на типа, като го заместват с ключовата дума **var**. С цел по-лесно разбиране ние ще изписваме типа при декларацията на променливите.

Сега ще преминем към имплементацията на всяка от горепосочените точки.

Прочитане на входните данни

Преди да преминем към самото прочитане на входните данни трябва да си **декларираме променливите**, в които ще ги съхраняваме. Това означава да им изберем подходящ тип данни и подходящи имена.

В променливата **n** ще съхраняваме броя на числата, които ще четем от конзолата. Избираме **тип int**, защото в условието е упоменато, че **n е цяло число** в диапазона от 1 до 1000. За променливите, в които ще пазим процентите, избираме **тип**

double, тъй като се очаква те **не винаги да са цели числа**. Допълнително си декларираме и променливите **cntP1**, **cntP2** и т.н., в които ще пазим броя на числата от съответната група, като за тях отново избираме **тип int**.

```
int n;

//Променливи, в които ще запазим
//процентното разделение на отделните групи
double p1Percentage = 0;
double p2Percentage = 0;
double p3Percentage = 0;
double p4Percentage = 0;
double p5Percentage = 0;

//Променливи, пазещи броя числа по групи
int cntP1 = 0;
int cntP2 = 0;
int cntP3 = 0;
int cntP4 = 0;
int cntP5 = 0;
```

След като сме си декларирали нужните променливи, можем да пристъпим към прочитането на числото **n** от конзолата:

```
n = int.Parse(Console.ReadLine());
```

Обработка на входните данни

За да прочетем и разпределим всяко число в съответната му група, ще си послужим с **for цикъл** от 0 до **n** (броя на числата). На всяка итерация от цикъла ще прочитаме и разпределяме **едно единствено** число (**currentNumber**) в съответната му група. За да определим дали едно число принадлежи към дадена група, правим **проверка за съответния ѝ диапазон**. Ако това е така ще увеличаваме броя на числата в тази група (**cntP1**, **cntP2** и т.н.) с 1.

Ето как можем да реализираме описаната логика за пребояване на входните числа по групи:

```
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++;
    }
}

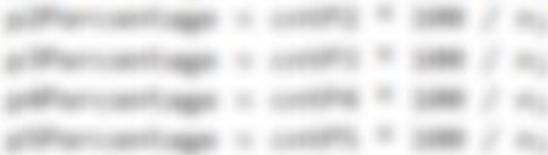
```

След като сме определили колко числа има във всяка група, можем да преминем към изчисляването на процентите, което е и главна цел на задачата. За това ще използваме следната формула:

$$(\text{процент на група}) = (\text{брой числа в група}) * 100 / (\text{брой на всички числа})$$

Тази формула в програмния код изглежда по следния начин:

```
p1Percentage = cntP1 * 100 / n;
```



Ако разделим на **100** (число тип **int**) вместо на **100.0** (число тип **double**), ще се извърши така нареченото **целочислено деление** и в променливата ще се запази само цялата част от делението, а това не е желания от нас резултат. Например: $5 / 2 = 2$, а $5 / 2.0 = 2.5$. Имайки това предвид, формулата за първата променлива ще изглежда така:

```
p1Percentage = cntP1 * 100.0 / n;
//Добавете формулите и за останалите променливи
```

За да стане още по-ясно какво се случва, нека разгледаме следния пример:

| Вход | Изход |
|------|--------|
| 3 | 66.67% |
| 1 | 0.00% |
| 2 | 0.00% |
| 999 | 33.33% |

В случая **n = 3**. За цикъла имаме:

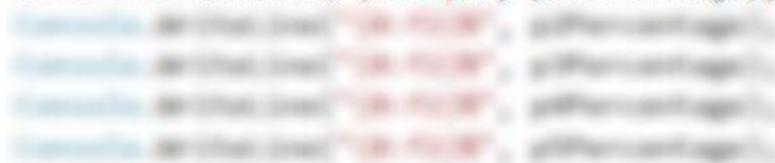
- **i = 0** – прочитаме числото 1, което е по-малко от 200 и попада в първата група (**p1**), увеличаваме брояча на групата (**cntP1**) с 1.
- **i = 1** – прочитаме числото 2, което отново попада в първата група (**p1**) и увеличаваме брояча ѝ (**cntP1**) отново с 1.
- **i = 2** – прочитаме числото 999, което попада в последната група (**p5**), защото е по-голямо от 800, и увеличаваме брояча на групата (**cntP5**) с 1.

След прочитането на числата в група **p1** имаме 2 числа, а в **p5** имаме 1 число. В другите групи **нямаме числа**. Като приложим гореспоменатата формула, изчисляваме процентите на всяка група. Ако във формулата умножим по **100**, вместо по **100.0** ще получим за група **p1** 66%, а за група **p5** – 33% (няма да има дробна част).

Извеждане на краен резултат

Остава само да отпечатаме получените резултати. В условието е казано, че процентите трябва да са **с точност две цифри след десетичната точка**. Това ще постигнем, като след placeholder-а изпишем “**:f2**”:

```
Console.WriteLine("{0:f2}%", p1Percentage);
```



Тестване в Judge системата

Тествайте решението си: <https://judge.softuni.bg/Contests/Practice/Index/511#0>.

Задача: умната Лили

Лили вече е на **N години**. За всеки свой **рожден ден** тя получава подарък. За **нечетните** рождения дни (1, 3, 5...n) получава **играчки**, а за всеки **четен** (2, 4, 6...n) получава **pari**. За **втория рожден ден** получава **10.00 лв.**, като **сумата се увеличава с 10.00 лв.** за всеки следващ четен рожден ден (2 -> 10, 4 -> 20, 6 -> 30...и т.н.). През годините Лили тайно е спестявала парите. **Братът** на Лили, в годините, които тя получава пари, взима по **1.00 лев** от тях. Лили продала **играчките**, получени през

годините, всяка за P лева и добавила сумата към спестените пари. С парите искала да си купи пералня за X лева. Напишете програма, която да пресмята **колко пари** е събрала и дали ѝ стигат да си купи пералня.

Входни данни

От конзолата се прочитат **3 числа**, всяко на отделен ред:

- Възрастта на Лили – **цяло число** в интервала [1...77].
- Цената на пералнята – **число** в интервала [1.00...10 000.00].
- Единична цена на играчка – **цяло число** в интервала [0...40].

Изходни данни

Да се отпечата на конзолата един ред:

- Ако парите на Лили са достатъчни:
 - “Yes! {N}” – където N е остатъка пари след покупката
- Ако парите не са достатъчни:
 - “No! {M}” – където M е сумата, която не достига
- Числата N и M трябва да са **форматирани до втория знак след десетичната точка**.

Примерен вход и изход

| Вход | Изход | Коментари |
|--------------------|------------|--|
| 10 170.00 6 | Yes! 5.00 | Първи рожден ден получава играчка ; 2ри -> 10 лв.; Зти -> играчка; 4ти -> $10 + 10 = 20$ лв.; 5ти -> играчка; 6ти -> $20 + 10 = 30$ лв.; 7ми -> играчка; 8ми -> $30 + 10 = 40$ лв.; 9ти -> играчка; 10ти -> $40 + 10 = 50$ лв. Спестила е -> $10 + 20 + 30 + 40 + 50 = 150$ лв.. Продала е 5 играчки по 6 лв. = 30 лв.. Брат ѝ взел 5 пъти по 1 лев = 5 лв. Остават -> $150 + 30 - 5 = 175$ лв. $175 \geq 170$ (цената на пералнята) Успяла е да я купи и са ѝ останали $175 - 170 = 5$ лв. |
| 21 1570.98 3 | No! 997.98 | Спестила е 550 лв.. Продала е 11 играчки по 3 лв. = 33 лв. Брат ѝ взимал 10 години по 1 лев = 10 лв. Останали $550 + 33 - 10 = 573$ лв. $573 < 1570.98$ – не е успяла да купи пералня. Не ѝ достигат $1570.98 - 573 = 997.98$ лв. |

Насоки и подсказки

Решението на тази задача, подобно на предходната, също можем да разделим мислено на три части – **прочитане** на входните данни, **обработката** им и **извеждане** на резултат.

```
int age = int.Parse(Console.ReadLine());
double priceOfWashingMachine =
int presentPrice =
```

Отново започваме с избора на подходящи **типове данни** и имена на променливите. За годините на Лили (**age**) и единичната цена на играчката (**presentPrice**) по условие е дадено, че ще са **цели числа**. Затова ще използваме типа **int**. За цената на пералнята (**priceOfWashingMachine**) знаем, че е **дробно число** и избираме **double**. Разбира се, можем да пропуснем изричното уточняване на типа, като вместо това употребим **var**. В кода по-горе **декларираме** и **инициализираме** (присвояваме стойност) променливите.

```
int numberofToys = 0;
int savedMoney = 0;
int moneyForBirthday = 10;
```

За да решим задачата, ще се нуждаем от няколко помощни променливи – за **броя на играчките** (**numberofToys**), за **спестените пари** (**savedMoney**) и за **парите, получени на всеки рожден ден** (**moneyForBirthday**). В началото присвояваме на **moneyForBirthday** първоначална стойност 10, тъй като по условие е дадено, че първата сума, която Лили получава, е 10 лв.

С **for цикъл** преминаваме през всеки рожден ден на Лили.

Когато водещата променлива е **четно число**, това означава, че Лили е **получила пари** и съответно прибавяме тези пари към общите ѝ спестявания. Едновременно с това **изваждаме по 1 лев** – парите, които брат ѝ взема. След това **увеличаваме** стойността на променливата **moneyForBirthday**, т.е. увеличаваме с 10 сумата, която тя ще получи на следващия си рожден ден.

Обратно, когато водещата променлива е **нечетно число**, увеличаваме броя на **играчките**.

Проверката за четност осъществяваме чрез **деление с остатък (%)** на 2 – когато остатъкът е 0, числото е четно, а при остатък 1 – нечетно.

```
for (int currentYear = 1; currentYear <= age; currentYear++)
{
    if (currentYear % 2 == 0)
    {
        savedMoney += (moneyForBirthday - 1);
        moneyForBirthday += 10;
    }
}
```

```

    }
else
{
    numberOfToys++;
}
}

```

Към спестяванията на Лили прибавяме и парите от продадените играчки:

```
savedMoney += numberOfToys * presentPrice;
```

Накрая остава да отпечатаме получените резултати, като се съобразим с форматирането, указано в условието, т.е. сумата трябва да е **закръглена до две цифри** след десетичния знак:

```
Console.WriteLine(savedMoney >= priceOfWashingMachine ?
        $"Yes! { (savedMoney - priceOfWashingMachine):0.00}"
        : $"No! { (priceOfWashingMachine - savedMoney):0.00}");
```

В случая избрахме да използваме **условния оператор (?:)** (наричан още тернарен оператор), тъй като записът е по-кратък. Синтаксисът му е следният:

операнд1 ? операнд2 : операнд3

Първият операнд трябва да е от булев тип (да връща **true/false**). Ако **операнд1** върне стойност **true**, то ще се изпълни **операнд2**, а ако върне **false** – **операнд3**. В нашия случай проверяваме дали **събрани пари** от Лили стигат за една пералня. Ако те са повече или равни на цената на пералнята, то проверката **savedMoney >= priceOfWashingMachine** ще върне **true** и ще се отпечата „**Yes! ...**“, а ако е по-малко – резултатът ще е **false** и ще се отпечата „**No! ...**“. Разбира се, вместо условния оператор, можем да използваме и **if** проверки.

Повече за условния тернарен оператор ?: <https://www.dotnetperls.com/ternary>, <https://msdn.microsoft.com/library/ty67wk28.aspx>.

Тестване в Judge системата

Тествайте решението си: <https://judge.softuni.bg/Contests/Practice/Index/511#1>.

Задача: завръщане в миналото

Иванчо е на **18 години** и получава наследство, което се състои от **X** **сума пари** и **машина на времето**. Той решава **да се върне до 1800 година**, но не знае **дали парите ще са достатъчни**, за да живее без да работи. Напишете **програма**, която пресмята **дали Иванчо ще има достатъчно пари**, за да не се налага да работи **до дадена година** включително. Като приемем, че **за всяка четна** (1800, 1802 и т.н.) **година ще харчи 12 000** **долара**. За **всяка нечетна** (1801, 1803 и т.н.) ще харчи **12 000 + 50 * [годините, които е навършил през дадената година]**.

Входни данни

Входът се чете от конзолата и съдържа точно 2 реда:

- **Наследените пари** – реално число в интервала [1.00...1 000 000.00].
- **Годината, до която трябва да живее (включително)** – цяло число в интервала [1801...1900].

Изходни данни

Да се отпечата на конзолата един ред. Сумата трябва да е форматирана с два знака след десетичния знак:

- Ако парите са достатъчно:
 - „Yes! He will live a carefree life and will have {N} dollars left.“ – където N са парите, които ще му останат.
- Ако парите НЕ са достатъчно:
 - „He will need {M} dollars to survive.“ – където M е сумата, която НЕ достига.

Примерен вход и изход

| Вход | Изход | Обяснения |
|-------------------|--|---|
| 50000 1802 | Yes! He will live a carefree life and will have 13050.00 dollars left. | 1800 → четна → Харчи 12000 долара → Остават $50000 - 12000 = 38000$ 1801 → нечетна → Харчи 12000 + $19 * 50 = 12950$ → Остават $38000 - 12950 = 25050$ 1802 → четна → Харчи 12000 → Остават $25050 - 12000 = 13050$ |
| 100000.15 1808 | He will need 12399.85 dollars to survive. | 1800 → четна → Остават $100000.15 - 12000 = 88000.15$ 1801 → нечетна → Остават $88000.15 - 12950 = 75050.15$... 1808 → четна → $-399.85 - 12000 = -12399.85$ 12399.85 не достигат |

Насоки и подсказки

Методът за решаване на тази задача не е по-различен от тези на предходните, затова започваме **деклариране и инициализиране** на нужните променливи:

```
double heritage = ...;
int yearToLive = ...;
int years = 18;
```

В условието е казано, че годините на Иванчо са 18, ето защо при декларацията на променливата **years** ѝ задаваме начална стойност **18**. Другите променливи прочитаме от конзолата.

```
for (int currentYear = 1800; currentYear <= yearToLive; currentYear++)
{
    if (currentYear % 2 == 0)
    {
        heritage -= 12000;
    }
    else
    {
        heritage -= (12000 + 50 * years);
    }
    years++;
}
```

С помощта на **for** цикъл ще обходим всички години. Започваме от 1800 – годината, в която Иванчо се връща, и стигаме до годината, до която той трябва да живее. В цикъла проверяваме дали текущата година е **четна** или **нечетна**. Проверката за четност осъществяваме чрез **деление с остатък (%)** на 2. Ако годината е **четна**, изваждаме от наследството (**heritage**) 12000, а ако е **нечетна**, изваждаме от наследството (**heritage**) $12000 + 50 * (\text{годините на Иванчо})$.

Накрая остава да отпечатаме резултатите, като за целта правим проверка дали наследството (**heritage**) му е било достатъчно да живее без да работи или не. Ако наследството (**heritage**) е положително число, отпечатваме: „**Yes! He will live a carefree life and will have {N} dollars left.**“, а ако е отрицателно число: „**He will need {M} dollars to survive.**“. Не забравяме да форматираме сумата до два знака след десетичната точка.

Hint: Обмислете използването на функцията **Math.Abs(...)** при отпечатване на изхода, когато наследството е недостатъчно.

Тестване в Judge системата

Тествайте решението си: <https://judge.softuni.bg/Contests/Practice/Index/511#2>.

Задача: болница

За даден период от време, всеки ден в болницата пристигат пациенти за преглед. Тя разполага **първоначално** със **7 лекари**. Всеки лекар може да преглежда **само по един пациент на ден**, но понякога има недостиг на лекари, затова **останалите пациенти се изпращат в други болници**. Всеки **трети ден** болницата прави изчисления и ако броят на непрегледаните пациенти е **по-голям от броя на прегледаните**, се назначава **още един лекар**. Като назначаването става преди да започне приемът на пациенти за деня.

Напишете програма, която изчислява **за дадения период** броя на прегледаните и непрегледаните пациенти.

Входни данни

Входът се чете от конзолата и съдържа:

- На първия ред – **периода**, за който трябва да направите изчисления. **Цяло число** в интервала [1...1000].
- На следващите редове (равни на броя на дните) – **броя пациенти**, които пристигат за преглед за **текущия ден**. Цяло число в интервала [0...10 000].

Изходни данни

Да се отпечатат на конзолата **2 реда**:

- На първия ред: "Treated patients: {брой прегледани пациенти}."
- На втория ред: "Untreated patients: {брой непрегледани пациенти}."

Примерен вход и изход

| Вход | Изход | Обяснения |
|------------------------|--|---|
| 4 7 27 9 1 | Treated patients: 23. Untreated patients: 21. | <p>1 ден: 7 прегледани и 0 непрегледани пациенти за деня</p> <p>2 ден: 7 прегледани и 20 непрегледани пациенти за деня</p> <p>3 ден: До момента прегледаните пациенти са общо 14, а непрегледаните – 20 –> Назначава се нов лекар –> 8 прегледани и 1 непрегледан пациент за деня</p> <p>4 ден: 1 прегледан и 0 непрегледани пациенти за деня</p> <p>Общо: 23 прегледани и 21 непрегледани пациенти.</p> |

| Вход | Изход | Вход | Изход |
|------|--|------|---|
| 6 | Treated patients: 40. Untreated patients: 87. | 3 | Treated patients: 21. Untreated patients: 0. |
| 25 | | 7 | |
| 25 | | 7 | |
| 25 | | 7 | |
| 25 | | | |
| 2 | | | |

Насоки и подсказки

Отново започваме, като си **декларираме и инициализираме** нужните променливи:

```
int period =
```

```
int treatedPatients = 0;
int untreatedPatients = 0;
int countOfDoctors = 7;
```

Периодът, за който трябва да направим изчисленията, прочитаме от конзолата и запазваме в променливата **period**. Ще се нуждаем и от няколко помощни променливи: броя на излекуваните пациенти (**treatedPatients**), броя на неизлекуваните пациенти (**untreatedPatients**) и броя на докторите (**countOfDoctors**), който първоначално е 7.

С помощта на **for цикъл** обхождаме всички дни в дадения период (**period**). За всеки ден прочитаме от конзолата броя на пациентите (**currentPatients**). Увеличаването на докторите по условие може да стане **всеки трети ден**, НО само ако броят на непрегледаните пациенти е **по-голям** от броя на прегледаните. За тази цел проверяваме дали денят е трети – чрез аритметичния оператор за деление с остатък (%): **day % 3 == 0**.

Например:

- Ако денят е **трети**, остатъкът от делението на 3 ще бъде 0 (**3 % 3 = 0**) и проверката **day % 3 == 0** ще върне **true**.
- Ако денят е **втори**, остатъкът от делението на 3 ще бъде 2 (**2 % 3 = 2**) и проверката ще върне **false**.
- Ако денят е **четвърти**, остатъкът от делението ще бъде 1 (**4 % 3 = 1**) и проверката отново ще върне **false**.

Ако проверката **day % 3 == 0** върне **true**, ще се провери дали и броят на неизлекуваните пациенти е по-голям от този на излекуваните: **untreatedPatients > treatedPatients**. Ако резултатът отново е **true**, тогава ще се увеличи броят на лекарите (**countOfDoctors**).

След това проверяваме броя на пациентите за деня (**currentPatients**) дали е по-голям от броя на докторите (**countOfDoctors**). Ако броят на пациентите е **по-голям**:

- Увеличаваме стойността на променливата **treatedPatients** с броя на докторите (**countOfDoctors**).
- Увеличаваме стойността на променливата **untreatedPatients** с броя на останалите пациенти, който изчисляваме, като от всички пациенти извадим броя на докторите (**currentPatients - countOfDoctors**).

Ако броят на пациентите **не е по-голям**, увеличаваме само променливата **treatedPatients** с броя на пациентите за деня (**currentPatients**).

Ето и примерна реализация на описания алгоритъм:

```
for (int day = 1; day <= period; day++)
{
    var currentPatients = ... // Read input here

    if ((day % 3 == 0) && (untreatedPatients > treatedPatients))
    {
        countOfDoctors++;
    }

    if (currentPatients > countOfDoctors)
    {
        treatedPatients += countOfDoctors;
        untreatedPatients += currentPatients - countOfDoctors;
    }
    else
    {
        treatedPatients += currentPatients;
    }
}
```

Накрая остава само **да отпечатаме** броя на излекуваните и броя на неизлекуваните пациенти и приключваме с тази задача. Реализирайте отпечатването сами.

Тестване в Judge системата

Тествайте решението си: <https://judge.softuni.bg/Contests/Practice/Index/511#3>.

Задача: деление без остатък

Дадени са n цели числа в интервала [1...1000]. От тях някакъв процент $p1$ се делят без остатък на 2, процент $p2$ се делят без остатък на 3, процент $p3$ се делят без остатък на 4. Да се напише програма, която изчислява и отпечатва процентите $p1$, $p2$ и $p3$. Пример: имаме $n = 10$ числа: 680, 2, 600, 200, 800, 799, 199, 46, 128, 65. Получаваме следното разпределение и визуализация:

| Деление без остатък на: | Числа | Брой | Процент |
|-------------------------|-----------------------------------|------|---------------------------------|
| 2 | 680, 2, 600, 200, 800, 46, 128 | 7 | $p1 = (7 / 10) * 100 = 70.00\%$ |
| 3 | 600 | 1 | $p2 = (1 / 10) * 100 = 10.00\%$ |
| 4 | 680, 600, 200, 800, 128 | 5 | $p3 = (5 / 10) * 100 = 50.00\%$ |

Входни данни

На първия ред от входа стои цялото число n ($1 \leq n \leq 1000$) – брой числа. На следващите n реда стои по едно цяло число в интервала [1...1000] – числата, които да бъдат проверени на колко се делят.

Изходни данни

Да се отпечатат на конзолата **3 реда**, всеки от които съдържа процент между 0% и 100%, с точност две цифри след десетичния знак, например 66.67%, 57.14%.

- На **първия ред** – процентът на числата, които **се делят на 2**.
- На **втория ред** – процентът на числата, които **се делят на 3**.
- На **третия ред** – процентът на числата, които **се делят на 4**.

Примерен вход и изход

| Вход | Изход | Вход | Изход | Вход | Изход |
|------|--------|------|---------|------|---------|
| 10 | 70.00% | 3 | 33.33% | 1 | 100.00% |
| 680 | 10.00% | 3 | 100.00% | 12 | 100.00% |
| 2 | 50.00% | 6 | 0.00% | 6 | 100.00% |
| 600 | | 9 | | 9 | |
| 200 | | | | | |
| 800 | | | | | |
| 799 | | | | | |
| 199 | | | | | |
| 46 | | | | | |
| 128 | | | | | |
| 65 | | | | | |

Насоки и подсказки

За тази и следващата задача ще трябва сами да напишете програмния код, следвайки дадените напътствия.

Програмата, която решава текущия проблем, е аналогична на тази от задача **Хистограма**, която разгледахме по-горе. Затова можем да започнем с декларацията на нужните ни променливи: Примерни имена на променливи може да са: **n** – брой на числата (който трябва да прочетем от конзолата) и **divisibleBy2**, **divisibleBy3**, **divisibleBy4** – помощни променливи, пазещи броя на числата от съответната група.

За да прочетем и разпределим всяко число в съответната му група, ще трябва да завъртим **for цикъл** от **0** до **n** (броя на числата). Всяка итерация на цикъла трябва да прочита и разпределя **едно единствено число**. Различното тук е, че **едно число може да попадне в няколко групи едновременно**, затова трябва да направим **три отделни if проверки за всяко число** – съответно дали се дели на 2, 3 и 4 (**if-else** конструкция в този случай няма да ни свърши работа, защото след като намери съвпадение се прекъсва по-нататъшното проверяване на условията) и да увеличим стойността на променливата, която пази броя на числата в съответната група.

Накрая трябва да отпечатате получените резултати, като спазвате посочения формат в условието.

Тестване в Judge системата

Тествайте решението си: <https://judge.softuni.bg/Contests/Practice/Index/511#4>.

Задача: логистика

Отговаряте за логистиката на различни товари. В **зависимост от теглото** на всеки товар е нужно **различно превозно средство** и струва различна цена на тон:

- До 3 тона – **микробус** (200 лева на тон).
- Над 3 и до 11 тона – **камион** (175 лева на тон).
- Над 11 тона – **влак** (120 лева на тон).

Вашата задача е да изчислите **средната цена на тон** превозен товар, както и **колко процента от товара** се превозват с **всяко превозно средство**.

Входни данни

От конзолата се четат **поредица от числа**, всяко на отделен ред:

- **Първи ред:** брой на товарите за превоз – **цяло число** в интервала [1...1000].
- На всеки следващ ред се подава **тонажът на поредния товар** – **цяло число** в интервала [1...1000].

Изходни данни

Да се отпечатат на конзолата 4 реда, както следва:

- Ред #1 – средната цена на тон превозен товар (закръглена до втория знак след десетичната точка).
- Ред #2 – процентът товар, превозван с **микробус** (между 0.00% и 100.00%, закръглен до втория знак след десетичната точка).
- Ред #3 – процентът товар, превозвани с **камион** (между 0.00% и 100.00%).
- Ред #4 – процентът товар, превозвани с **влак** (между 0.00% и 100.00%).

Примерен вход и изход

| Вход | Изход | Обяснения |
|------|--------|--|
| 4 | 143.80 | С микробус се превозват два от товарите 1 + 3 , общо 4 тона. |
| 1 | 16.00% | С камион се превозва един от товарите: 5 тона. |
| 5 | 20.00% | С влак се превозва един от товарите: 16 тона. |
| 16 | 64.00% | Сумата от всички товари е: 1 + 5 + 16 + 3 = 25 тона. Процент товар с микробус : $4/25 * 100 = 16.00\%$ Процент товар с камион : $5/25 * 100 = 20.00\%$ Процент товар с влак : $16/25 * 100 = 64.00\%$ Средна цена на тон превозен товар: $(4 * 200 + 5 * 175 + 16 * 120) / 25 = 143.80$ |
| 3 | | |

| Вход | Изход | Вход | Изход |
|------|--------|------|--------|
| 5 | 149.38 | 4 | 120.35 |
| 2 | 7.50% | 53 | 0.00% |
| 10 | 42.50% | 7 | 0.63% |
| 20 | 50.00% | 56 | 99.37% |
| 1 | | 999 | |
| 7 | | | |

Насоки и подсказки

Първо ще прочетем теглото на всеки товар и ще сумираме колко тона се превозят съответно с **микробус**, **камион** и **влак** и ще изчислим и **общите тонове** превозени товари. Ще пресметнем **цените за всеки вид транспорт** според превозените тонове и **общата цена**. Накрая ще пресметнем и отпечатаме **общата средна цена на тон** и каква част от товара е превозена с всеки вид транспорт процентно.

Декларираме си нужните променливи, например: **countOfLoads** – броя на товарите за превоз (прочитаме ги от конзолата), **sumOfTons** – сумата от тонажа на всички товари, **microbusTons**, **truckTons**, **trainTons** – променливи, пазещи сумата от тонажа на товарите, превозвани съответно с микробус, камион и влак.

Ще ни трябва **for** цикъл от **0** до **countOfLoads-1**, за да обходим всички товари. За всеки товар прочитаме теглото му (в тонове) от конзолата и го запазваме в променлива, например **tons**. Прибавяме към сумата от тонажа на всички товари (**sumOfTons**) теглото на текущия товар (**tons**). След като сме прочели теглото на текущия товар, трябва да определим кое превозно средство ще се ползва за него (микробус, камион или влак). За целта ще ни трябват **if-else** проверки:

- Ако стойността на променливата **tons** е по-малка от **3**, увеличаваме стойността на променливата **microbusTons** със стойността на **tons**:

microbusTons += tons;

- Иначе, ако стойността **tons** е до **11**, увеличаваме **truckTons** с **tons**.
- Ако **tons** е повече от **11**, увеличаваме **trainTons** с **tons**.

Преди да отпечатаме изхода трябва да изчислим процента на тоновете, превозвани с всяко превозно средство и средната цена на тон. За средната цена на тон ще си декларираме още една помощна променлива **totalPrice**, в която ще сумираме общата цена на всички превозвани товари (с микробус, камион и влак). Средната цена ще получим, разделяйки **totalPrice** на **sumOfTons**. Остава сами да изчислите процента на тоновете, превозвани с всяко превозно средство, и да отпечатате резултатите, спазвайки формата в условието.

Тестване в Judge системата

Тествайте решението си: <https://judge.softuni.bg/Contests/Practice/Index/511#5>.

Глава 6.1. Вложени цикли

В настоящата глава ще разгледаме **вложените цикли** и как да използваме **for** цикли за **чертане** на различни **фигурки** на конзолата, които се състоят от символи и знаци, разположени в редове и колони на конзолата. Ще използваме **единични** и **вложени цикли** (цикли един в друг), **изчисления** и **проверки**, за да отпечатваме на конзолата прости и не чак толкова прости фигурки по зададени размери.

Видео

Гледайте видео-урок по тази глава: <https://youtube.com/watch?v=x7zXRCpkebo>.

Пример: правоъгълник от 10 x 10 звездички

Да се начертате в конзолата правоъгълник от **10 x 10** звездички.

| Вход | Изход |
|--------|--|
| (няма) | ***** ***** ***** ***** ***** ***** ***** ***** ***** ***** |

Насоки и подсказки

```
for (var i = 1; i <= 10; i++)  
{  
    Console.WriteLine(new string('*', 10));  
}
```

Как работи примерът? Инициализира се **цикъл** с променлива **i = 1**, която се увеличава на всяка итерация на цикъла, докато е **по-малка или равна на 10**. Така кодът в тялото на цикъла се изпълнява **10 пъти**. В тялото на цикъла се печата на нов ред в конзолата **new string('*', 10)**, което създава низ от 10 звездички.

Тестване в Judge системата

Тествайте решението си: <https://judge.softuni.bg/Contests/Practice/Index/512#0>.

Пример: правоъгълник от N x N звездички

Да се напише програма, която въвежда цяло положително число **n** и печата на конзолата **правоъгълник от N x N звездички**.

| Вход | Изход | Вход | Изход | Вход | Изход |
|------|----------|------|-------------------|------|------------------------------|
| 2 | ** ** | 3 | *** *** *** | 4 | **** **** **** **** |
| | | | | | |

Насоки и подсказки

```
int n = int.Parse(Console.ReadLine());
for (int i = 1; i <= n; i++)
{
    Console.WriteLine(
        new string('*', n));
}
```

Тестване в Judge системата

Тествайте решението си: <https://judge.softuni.bg/Contests/Practice/Index/512#1>.

Вложени цикли

Вложените цикли представляват конструкция, при която **в тялото на един цикъл** (външен) се изпълнява **друг цикъл** (вътрешен). За всяко завъртане на външния цикъл, вътрешният се извърта **целият**. Това се случва по следния начин:

- При стартиране на изпълнение на вложени цикли първо **стартира външният цикъл**: извършва се **инициализация** на неговата управляваща променлива и след проверка за край на цикъла, се изпълнява кодът в тялото му.
- След това се **изпълнява вътрешният цикъл**. Извършва се инициализация на началната стойност на управляващата му променлива, прави се проверка за край на цикъла и се изпълнява кодът в тялото му.
- При достигане на зададената стойност за **край на вътрешния цикъл**, програмата се връща една стъпка нагоре и се продължава започналото изпълнение предходния (външния) цикъл. Променя се с една стъпка управляващата променлива за външния цикъл, проверява се дали условието за край е изпълнено и **започва ново изпълнение на вложения (вътрешния) цикъл**.
- Това се повтаря докато променливата на външния цикъл достигне условието за **край на цикъла**.

Ето и един **пример**, с който нагледно да илюстрираме вложените цикли. Целта е да се отпечата отново правоъгълник от **n * n** звездички, като за всеки ред се извърта цикъл от **1** до **n**, а за всяка колона се извърта вложен цикъл от **1** до **n**:

```

var n = int.Parse(Console.ReadLine());
for (var r = 1; r <= n; r++)
{
    for (var c = 1; c <= n; c++)
    {
        Console.WriteLine("*");
    }
    Console.WriteLine();
}

```

Да разгледаме примера по-горе. След инициализацията на **първия** (външен) цикъл, започва да се изпълнява неговото **тяло**, което съдържа **втория** (вложен) цикъл. Той сам по себе си печата на един ред **n** на брой звездички. След като **вътрешният** цикъл **приключи** изпълнението си при първата итерация на външния, то след това **външният ще продължи**, т.е. ще отпечата един празен ред на конзолата. **След това** ще се извърши **обновяване** на променливата на **първия** цикъл и отново ще бъде изпълнен целият **втори** цикъл. Вътрешният цикъл ще се изпълни толкова пъти, колкото се изпълнява тялото на външния цикъл, в случая **n** пъти.

Пример: квадрат от звездички

Да се начертате на конзолата квадрат от $N \times N$ звездички:

| Вход | Изход | Вход | Изход | Вход | Изход |
|------|----------------------|------|--------------------------------|------|--|
| 2 | <pre> * * * * </pre> | 3 | <pre> * * * * * * * * * </pre> | 4 | <pre> * * * * * * * * * * * * * * * * </pre> |
| | | | | | |

Насоки и подсказки

Задачата е аналогична на предходната. Разликата тук е, че в тази трябва да обмислим как да печатаме интервал след звездичките по такъв начин, че да няма излишни интервали в началото или края.

```

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();
}
}
```

Тестване в Judge системата

Тествайте решението си: <https://judge.softuni.bg/Contests/Practice/Index/512#2>.

Пример: триъгълник от долари

Да се напише програма, която въвежда число **n** и печата **триъгълник от долари**.

| Вход | Изход | Вход | Изход |
|------|-------------------------|------|--|
| 3 | \$ \$ \$ \$ \$ \$ | 4 | \$ \$ \$ \$ \$ \$ \$ \$ \$ \$ |
| | | | |

Насоки и подсказки

Задачата е сходна с тези за рисуване на **правоъгълник** и **квадрат**. Отново ще използваме **вложени цикли**, но тук има **уловка**. Разликата е в това, че **броя на колонките**, които трябва да разпечатаме, зависят от **реда**, на който се намираме, а не от входното число **n**.

От примерните входни и изходни данни забелязваме, че **броят на долларите зависи** от това на кой **ред** се намираме към момента на печатането, т.е. 1 доллар означава първи ред, 3 долара означават трети ред и т.н.

Да разгледаме долния пример по-подробно. Виждаме, че **променливата** на **вложения цикъл** е обвързана с променливата на **външния**. По този начин нашата програма печата желания триъгълник.

Ето и **примерно решение** на задачата:

```
var n = int.Parse(Console.ReadLine());
for (var row = 1; row <= n; row++)
{
    Console.Write("$");
    for (var col = 1; col < row; col++)
    {
        Console.Write(" $");
    }
    Console.WriteLine();
}
```

Тестване в Judge системата

Тествайте решението си: <https://judge.softuni.bg/Contests/Practice/Index/512#3>.

Пример: квадратна рамка

Да се напише програма, която въвежда цяло положително число n и чертае на конзолата **квадратна рамка** с размер $n * n$.

| Вход | Изход | Вход | Изход | Вход | Изход |
|------|--|------|---|------|--|
| 4 | + - - + - - - - + - - + | 5 | + - - - + - - - - - - - - - + - - - + | 6 | + - - - - + - - - - - - - - - - - - - - - - + - - - - + |
| | | | | | |

Насоки и подсказки

Можем да решим задачата по следния начин:

- Четем от конзолата числото n .
- Отпечатваме **горната част**: първо знак **+**, после $n-2$ пъти **-** и накрая знак **+**.
- Отпечатваме **средната част**:
 - Печатаме $n-2$ реда като първо печатаме знак **|**, после $n-2$ пъти **-** и накрая **|**. Това можем да го постигнем с вложени цикли.
- Отпечатваме **долната част**: първо **+**, после $n-2$ пъти **-** и накрая **+**.

Ето и примерна имплементация на описаната идея, с вложени цикли:

```
int n = int.Parse(Console.ReadLine());

// Print the top row: + - - - +
Console.WriteLine("+");
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(" +");

```

Тестване в Judge системата

Тествайте решението си: <https://judge.softuni.bg/Contests/Practice/Index/512#4>.

Пример: ромбче от звездички

Да се напише програма, която въвежда цяло положително число n и печата ромбче от звездички с размер n .

| Вход | Изход | Вход | Изход | Вход | Изход | Вход | Изход |
|------|-------|------|-------|------|-----------|------|-----------|
| 1 | * | 2 | * * * | 3 | * * * * * | 4 | * * * * * |
| | | | | | | | |
| | | | | | | | |

Насоки и подсказки

За решението на тази задача е нужно да разделим мислено ромба на две части – горна, която включва и средния ред, и долната. За разпечатването на всяка една част ще използваме два отделни цикъла, като оставяме на читателя сам да намери зависимостта между n и променливите на циклите. За първия цикъл може да използваме следните насоки:

- Отпечатваме **n-row** интервала.

- Отпечатваме *.
- Отпечатваме **row-1** пъти *.

Ето и примерна имплементация на описаната идея:

```
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
```

Втората (долна) част ще разпечатаме по **аналогичен** начин, което отново оставяме на читателя да се опита да направи сам.

Тестване в Judge системата

Тествайте решението си: <https://judge.softuni.bg/Contests/Practice/Index/512#5>.

Пример: коледна елха

Да се напише програма, която въвежда число **n** ($1 \leq n \leq 100$) и печата коледна елха с височина **n+1**.

| Вход | Изход | Вход | Изход | Вход | Изход | Вход | Изход |
|------|-------|------|------------------|------|-------------------------------|------|--|
| 1 | * * | 2 | * * ** ** | 3 | * * ** ** *** *** | 4 | * * ** ** *** *** **** **** |

Насоки и подсказки

От примерите виждаме, че **елхата** може да бъде разделена на **три логически части**. **Първата** част са **звездинките** и **празните места** преди и след тях, **средната** част е **|**, а **последната** част са отново **звездинки**, като този път **празни места** има само **преди** тях. Разпечатването може да бъде постигнато само с **един цикъл** и конструктора **new string(...)**, който ще използваме един път за звездичките и един път за интервалите.

```

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.Write(spaces);
    Console.Write(stars);
    Console.Write(" | ");
    Console.Write(stars);
    Console.WriteLine(spaces);
}

```

Тестване в Judge системата

Тествайте решението си: <https://judge.softuni.bg/Contests/Practice/Index/512#6>.

Чертане на по-сложни фигури

Да разгледаме как можем да чертаем на конзолата фигури с по-сложна логика на конструиране, за които трябва повече да помислим преди да почнем да пишем.

Пример: слънчеви очила

Да се напише програма, която въвежда цяло число n ($3 \leq n \leq 100$) и печата слънчеви очила с размер $5 * n \times n$ като в примерите:

| Вход | Изход |
|------|---|
| 3 | ***** * **** *////* *////* ***** * ***** |

| Вход | Изход |
|------|---|
| 4 | ***** * ***** *////* *////* *////* *////* ***** * ***** |

| Вход | Изход |
|------|--|
| 5 | ***** * ***** *////* *////* *////* *////* *////* *////* ***** * ***** |

Насоки и подсказки

От примерите виждаме, че очилата могат да се разделят на **три части** – горна, средна и долну. По-долу е част от кода, с който задачата може да се реши.

При рисуването на горния и долния ред трябва да се отпечатат $2 * n$ звездички, n интервала и $2 * n$ звездички.

```
// Print the top part
Console.Write(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.Write(new string('*', 2 * n));
Console.Write(new string(' ', n));
Console.WriteLine(new string('*', 2 * n));
```

При печатането на **средната** част трябва да **проверим** дали редът е $(n-1) / 2 - 1$, тъй като от примерите е видно, че на този ред трябва да печатаме **вертикални чертички** вместо интервали.

```
// Print the middle part
for (int i = 0; i < n - 2; i++)
{
    // TODO: print *////////*
    if (i == (n - 1) / 2 - 1)
        Console.Write(new string('|', n));
    else
        Console.Write(new string(' ', n));
    // TODO: print *////////*
    Console.WriteLine();
}
```

Тестване в Judge системата

Тествайте решението си: <https://judge.softuni.bg/Contests/Practice/Index/512#7>.

Пример: къщичка

Да се напише програма, която въвежда число n ($2 \leq n \leq 100$) и печата **къщичка** с размери $n \times n$, точно като в примерите:

| Вход | Изход | Вход | Изход | Вход | Изход | Вход | Изход |
|------|--------|------|---------------------|------|----------------------------------|------|---|
| 2 | ** | 3 | - * - *** * | 4 | - ** - **** ** ** | 5 | - - * - - - - *** - - ***** *** *** |
| | | | | | | | |

Насоки и подсказки

Разбираме от условието на задачата, че къщата е с размер **n x n**. Това, което виждаме от примерните вход и изход, е че:

- Къщичката е разделена на 2 части: **покрив и основа**.
- Когато **n** е четно число, върхът на къщичката е "тъп".
- Когато **n** е нечетно число, **покривът** е с един ред по-голям от **основата**.



Покрив

- Съставен е от **звезди** и **тирета**.
- В най-високата си част има една или две звезди, спрямо това дали **n** е четно или нечетно, както и тирета.
- В най-ниската си част има много звезди и малко или никакви долни черти.
- С всеки един ред по-надолу, **звездите** се увеличават с 2, а **тиретата** намаляват с 2.

Основа

- Дълга е **n** на брой реда.
- Съставена е от **звезди** и **тирета**.
- Редовете представляват 2 **тирета** – по едно в началото и в края на реда, както и **звезди** между тиретата с дължина на низа **n - 2**.

Прочитаме **n** от конзолата и записваме стойността в променлива от тип **int**.

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



Много е важно да проверяваме дали са **валидни** входните данни! В тези задачи не е проблем директно да обръщаме прочетеното от конзолата в тип **int**, защото изрично е казано че ще получаваме **валидни** целочислени числа. Ако обаче правите по-сериозни приложения е добра практика да проверявате данните. Какво ще стане, ако вместо буквата "A" потребителя въведе число?

За да начертаем покрива, записваме колко ще е началният брой звезди в променлива **stars**:

- Ако **n** е четно число, ще са 2 броя.
- Ако е нечетно, ще е 1 брой.

```
var stars = 1;
if (n % 2 == 0)
{
    stars++;
}
```

Изчисляваме дължината на покрива. Тя е равна на половината от **n**. Резултата записваме в променливата **roofLength**.

```
var roofLength = (int)Math.Ceiling(n / 2f);
```

Важно е да се отбележи че, когато **n** е нечетно число, дължината на покрива е поголяма с един ред от тази на **основата**. В езика **C#**, когато два целочислени типа се делят и има остатък, то резултата ще е числото без остатъка. Пример:

```
int result = 3 / 2; // резултат 1
```

Ако искаме да закръглим нагоре, трябва да използваме метода **Math.Ceiling(...)**:
int result = (int)Math.Ceiling(3 / 2f); В този пример делението не е от 2 целочислени числа. "f" след число показва, че даденото число е от тип **float** (число с плаваща запетая). Резултатът от **3 / 2f** е **1.5f**. **Math.Ceiling...** закръгля делението нагоре. В нашият случай **1.5f** ще стане 2. **(int)** се използва, за да може да трансформираме типа обратно в **int**.

След като сме изчислили дължината на покрива, завъртаме цикъл от 0 до **roofLength**. На всяка итерация ще:

- Изчисляваме броя **тиreta**, които трябва да изрисуваме. Броят ще е равен на **(n - stars) / 2**. Записваме го в променлива **padding**.

```
var padding = (n - stars) / 2;
```

- Отпечатваме на конзолата: "тиreta" (**padding / 2** на брой пъти) + "звезди" (**stars** пъти) + "тиreta" (**padding / 2** пъти).

```
var line = new string('-', padding)
    + new string('*', stars)
    + new string('-', padding);
Console.WriteLine(line);
```

- Преди да свърши итерацията на цикъла добавяме 2 към **stars** (броя на звездите).

```
stars += 2;
```



Не е добра идея да правим събирания на много на брой символни низове по показания по-горе начин, защото това води до **проблеми със скоростта** (performance issues). За повече информация посетете: http://bg.wikipedia.org/wiki/%D0%9D%D0%B8%D0%B7#String_Builder

След като сме приключили с **покрива**, е време за **основата**. Тя е по-лесна за печатане, отколкото е покривът.

- Започваме с цикъл от **0** до **n** (изключено).
- Отпечатваме на конзолата: **| + * (n - 2** на брой пъти) **+ |**.

```
for (int i = 0; i < n / 2; i++)
{
    var line = "|" + new string('*', n - 2) + "|";
    Console.WriteLine(line);
}
```

Ако всичко сме написали както трябва, задачата ни е решена.

Тестване в Judge системата

Тествайте решението си: <https://judge.softuni.bg/Contests/Practice/Index/512#8>.

Пример: диамант

Да се напише програма, която въвежда цяло число **n** ($1 \leq n \leq 100$) и печата диамант с размер **n**, като в следните примери:

| Вход | Изход | Вход | Изход | Вход | Изход | Вход | Изход | Вход | Изход |
|------|-------|------|-------|------|-------------------|------|----------------------|------|---------------------------------------|
| 1 | * | 2 | ** | 3 | -*- *-* -*- | 4 | -**- *--* -**- | 5 | --*- -*-* *---* -*-* --*- |

Насоки и подсказки

Това, което знаем от условието на задачата, е че диамантът е с размер **n x n**.

От примерните вход и изход можем да си направим изводи, че всички редове съдържат точно по **n** символа и всички редове, с изключение на горните върхове, имат по **2 звезди**. Можем мислено да разделим диаманта на 2 части:

- **Горна** част. Тя започва от горният връх до средата.

- **Долна** част. Тя започва от реда след средата до най-долнния връх (включително).

Горна част

- Ако **n** е **нечетно**, то тя започва с **1 звезда**.
- Ако **n** е **четно**, то тя започва с **2 звезди**.
- С всеки ред надолу, звездите се отдалечават една от друга.
- Пространството между, преди и след **звездите** е запълнено с **тиreta**.

Долна част

- С всеки ред надолу, звездите се събират една с друга. Това означава, че пространството (**тиretata**) между тях намалява, а пространството (**тиretata**) отляво и отдясно се увеличава.
- В най-долната си част е с **1** или **2 звезди**, спрямо това дали **n** е четно или не.

Горна и долна част на диаманта

- На всеки ред звездите са заобиколени от външни **тиreta**, с изключение на средния ред.
- На всеки ред има пространство между двете **звезди**, с изключение на първия и последния ред (понякога **звездата е 1**).

Прочитаме стойността на **n** от конзолата и я записваме в променлива от тип **int**.

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

Започваме да чертаем горната част на диаманта. Първото трябва да изчислим началната стойност на външната бройка **тиreta leftRight** (тиretata от външната част на звездите). Тя е равна на **(n - 1) / 2**, закръглено надолу.

```
var leftRight = (n - 1) / 2;
```

След като сме изчислили **leftRight**, започваме да чертаем горната част на диаманта. Може да започнем, като завъртим **цикъл** от **0** до **n / 2 + 1** (закръглено надолу).

При всяка итерация на цикъла трябва да се изпълнят следните стъпки:

- Рисуваме по конзолата левите **тиreta** (с дължина **leftRight**) и веднага след тях първата **звезда**.

```
Console.WriteLine(new string('-', leftRight));  
Console.Write("*");
```

- Ще изчислим разстоянието между двете **звезди**. Може да го изчислим като извадим от **n** дължината на външните **тиreta**, както и числото 2 (бройката на **звездите**, т.е. очертанията на диаманта). Резултата от тази разлика записваме в променлива **mid**.

```
var mid = n - 2 * leftRight - 2;
```

- Ако **mid** е по-малко от 0, то тогава знаем, че на реда трябва да има 1 звезда. Ако е по-голямо или равно на 0, то тогава трябва да начертаем **тирета** с дължина **mid** и една **звезда** след тях.
- Рисуваме на конзолата десните външни **тирета** с дължина **leftRight**.

```
Console.WriteLine(new string('-', leftRight));
```

- В края на цикъла намаляваме **leftRight** с 1 (**звездите** се отдалечават).

Готови сме с горната част.

Рисуването на долната част е доста подобна на рисуването на горната част. Разликите са, че вместо да намаляваме **leftRight** с 1 към края на цикъла, ще увеличаваме **leftRight** с 1 в началото на цикъла. Също така, **цикълът ще е от 0 до $(n - 1) / 2$** .

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



Повторението на код се смята за лоша практика, защото кодът става доста труден за поддръжка. Нека си представим, че имаме парче код (напр. логиката за чертането на ред от диаманта) на още няколко места и решаваме да направим промяна. За целта би било необходимо да минем през всичките места и да направим промените. Нека си представим, че трябва да използвате код не 1, 2 или 3 пъти, а десетки пъти. Начин за справяне с този проблем е като се използват **методи**. Можете да потърсите допълнителна информация за тях в Интернет, както и по-късно в [глава "10" \(Методи\)](#).

Ако сме написали всичко коректно, задачата ни е решена.

Тестване в Judge системата

Тествайте решението си: <https://judge.softuni.bg/Contests/Practice/Index/512#9>.

Какво научихме от тази глава?

Запознахме се с конструктора **new string**:

```
string printMe = new string('*', 5);
```

Научихме се да чертаем фигури с вложени **for** цикли:

```
for (var r = 1; r <= 5; r++)
{
    Console.Write("*");
    for (var c = 1; c < 5; c++)
        Console.Write(" *");
```

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

Упражнения: чертане на фигурки в уеб среда

Сега, след като свикнахме с **вложените цикли** и как да ги използваме, за да чертаем фигури на конзолата, можем да се захванем с нещо още по-интересно: да видим как циклите могат да се използват за **чертане в уеб среда**. Ще направим уеб приложение, което визуализира числовой рейтинг (число от 0 до 100) със звездички. Такава визуализация се среща често в сайтове за електронна търговия, реюта на продукти, оценки на събития, рейтинг на приложения и други.

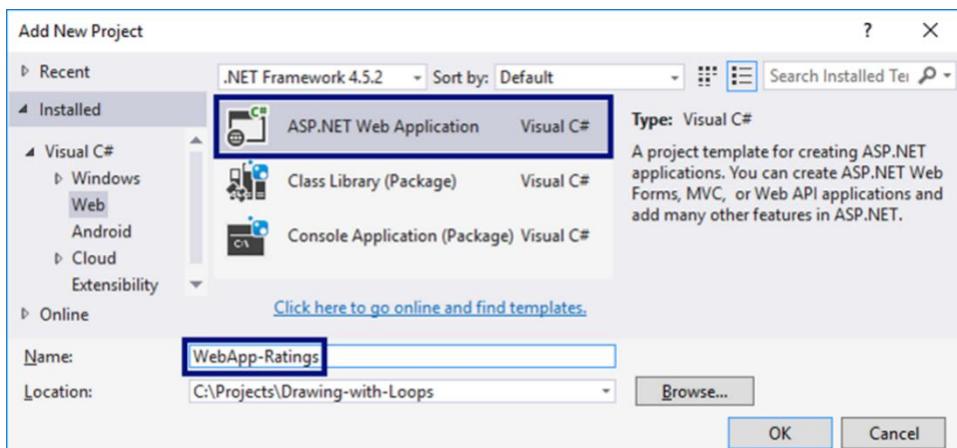
Не се притеснявайте, ако не разберете целия код, как е точно е направен и как точно работи проектът. Нормално е, сега се учим да пишем код, не сме стигнали до технологиите за уеб разработка. Ако имате трудности да си напишете проекта, следвайки описаните стъпки, гледайте видеото от началото на тази глава или питайте в СофтУни форума: <https://softuni.bg/forum>.

Задача: рейтинги – визуализация в уеб среда

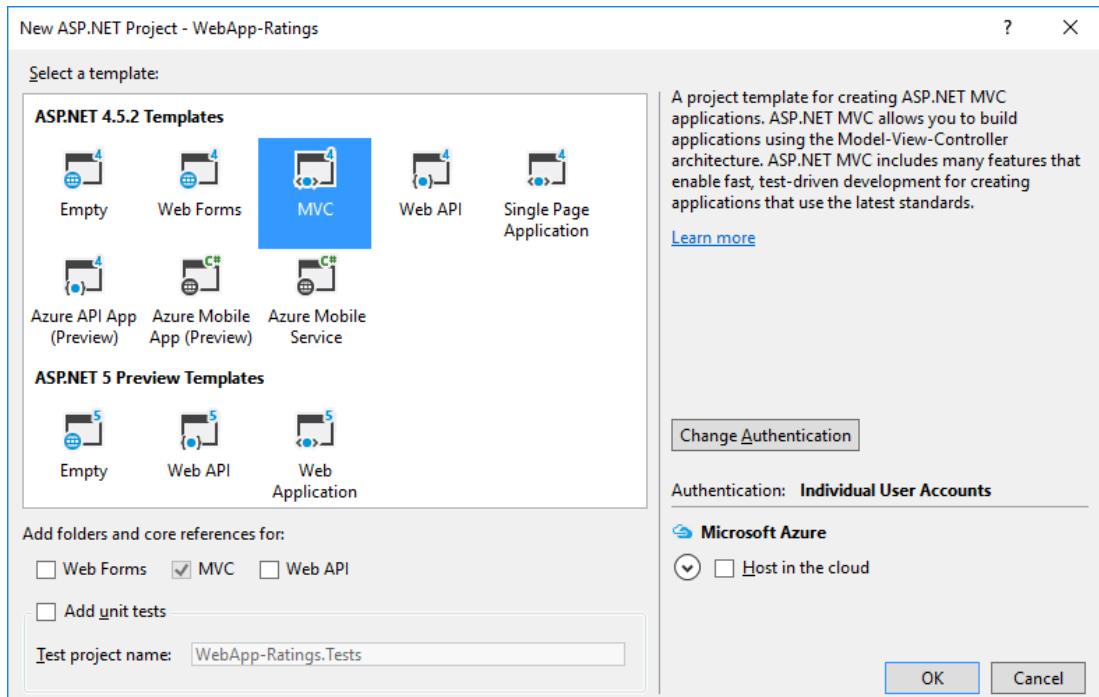
Да се разработи ASP.NET MVC уеб приложение за визуализация на рейтинг (число от 0 до 100). Чертаят се от 1 до 10 звездички (с половинки). Звездичките да се генерират с **for** цикъл.



Във Visual Studio създаваме ново ASP.NET MVC уеб приложение с език C#. Добавяме нов проект от [Solution Explorer] -> [Add] -> [New Project...]. Даваме смислено име, например "WebApp-Ratings".



Избираме тип на уеб приложението **MVC**.



Отваряме и редактираме файла **Views/Home/Index.cshtml**. Изтриваме всичко и въвеждаме следния код:

Rating

<form action="DrawRating">
 <input type="number" min="0" max="100" name="rating" value="@ViewBag.Rating" />
 <input type="submit" value="Rate" />
</form>

@Html.Raw(ViewBag.Stars)

The 'Solution Explorer' window shows the project structure with 'WebApp-Ratings' containing 'Properties', 'References', 'App_Data', 'App_Start', 'Content', 'Controllers', 'fonts', 'images', 'Models', 'Scripts', 'Views' (containing 'Account', 'Home' with 'About.cshtml' and 'Index.cshtml' selected, 'Manage', 'Shared', and '_ViewStart.cshtml')."/>

```

@{
    ViewBag.Title = "Rating";
}



# Rating



<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)

```

Този код създава уеб форма `<form>` с поле `"rating"` за въвеждане на число в интервала [0...100] и бутон [Draw] за изпращане на данните към сървъра.

Действието, което ще обработи данните, се назова **Home/DrawRatings**, което означава метод **DrawRatings** в контролер **Home**, който се намира във файла **HomeController.cs**. След формата се отпечатва съдържанието на **ViewBag.Stars**. Кодът, който ще се съдържа в него, ще бъде динамично генериран от контролера HTML с поредица от звездички.

Добавяме метод **DrawRatings** в контролера **HomeController**. Отваряме файла **Controllers/HomeController.cs** и добавяме следния код:

```
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' />";

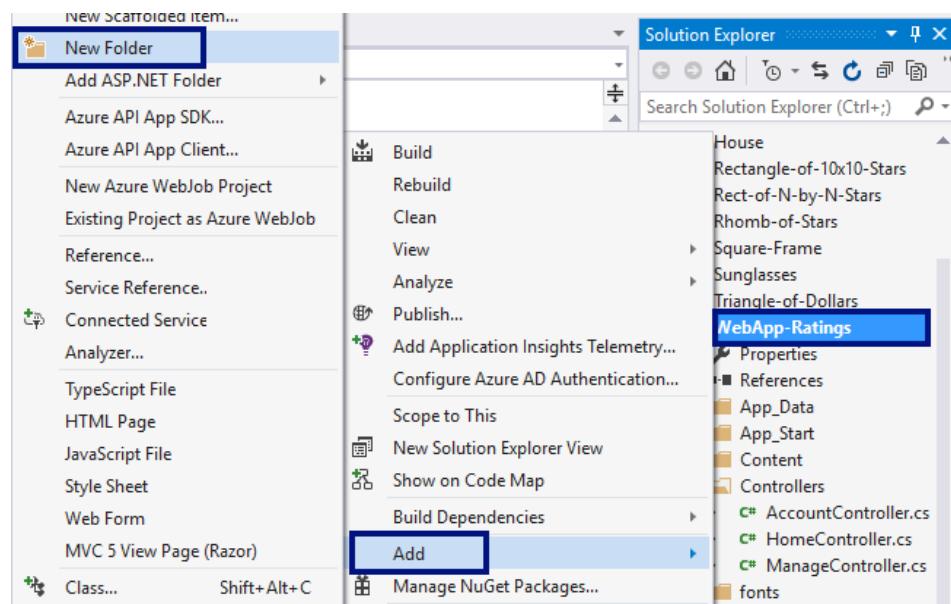
    for (int i = 0; i < emptyStars; i++)
        stars += "<img src='/images/empty-star.png' />";

    ViewBag.Stars = stars;
    ViewBag.Rating = rating;
    return View("Index");
}
```

Горният код взима въведеното число **rating**, прави малко пресмятания и изчислява броя **пълни звездички**, броя **празни звездички** и броя **половинки звездички**, след което генерира HTML код, който нареджа няколко картинки със звездички една след друга, за да сглоби от тях картинката с рейтинга.

Подгответият HTML код се записва във **ViewBag.Stars** за визуализация от изгледа **Index.cshtml**. Допълнително се запазва и изпратеният рейтинг (като число) във **ViewBag.Rating**, за да се зададе в полето за рейтинг в изгледа.

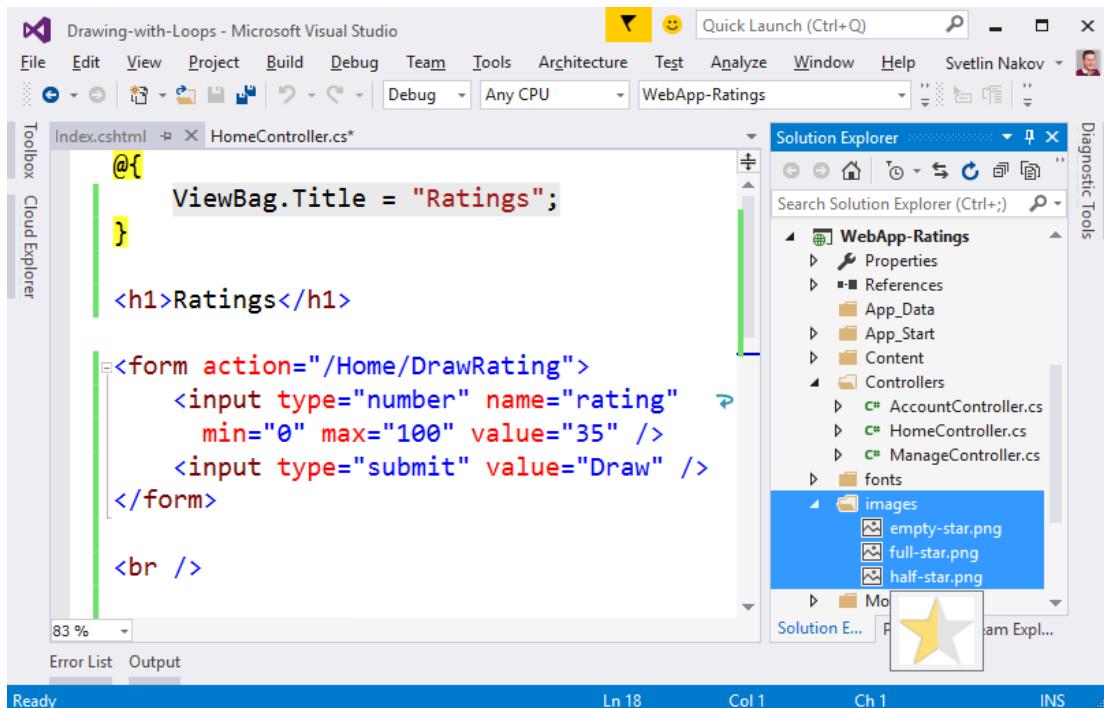
От [Solution Explorer] създаваме нова папка **images** в проекта:



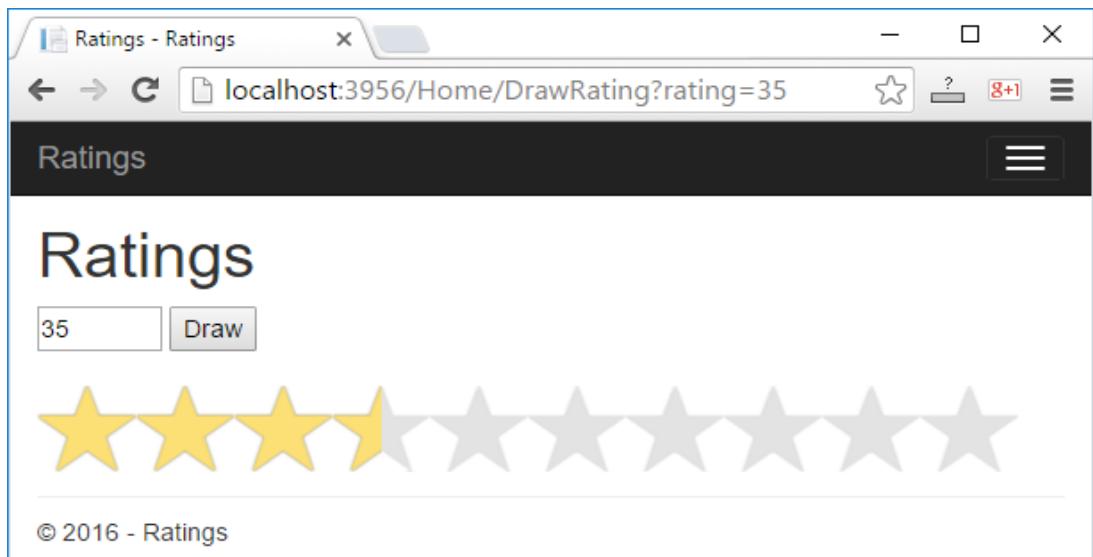
Сега добавяме **картинките със звездичките** (те са част от файловете със заданието за този проект и могат да бъдат свалени от хранилището на книгата в GitHub:

<https://github.com/SoftUni/Programming-Basics-Book-CSharp-BG/tree/master/assets/chapter-6-assets>

Копираме ги от Windows Explorer и ги поставяме в папката **images** в [Solution Explorer] във Visual Studio с copy/paste.



Стартираме проекта с [Ctrl+F5] и му се наслаждаваме:

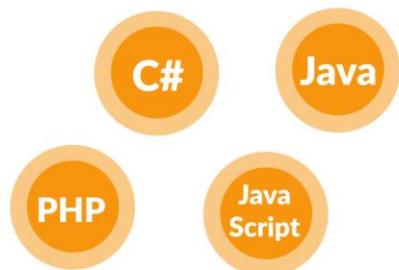


Ако имате проблеми с примерния проект по-горе, [гледайте видеото](#) в началото на тази глава. Там приложението е направено на живо стъпка по стъпка с много обяснения. Или питайте във [форума на СофтУни](https://softuni.bg/forum): <https://softuni.bg/forum>.

Качествено образование,
професия и работа за

Софтуерни инженери

- ✓ Безплатен старт за начинаещи - **присъствено и онлайн**
- ✓ Избор измежду **най-търсените професии** в софтуерната индустрия
- ✓ Съдействие за **кариерен старт**
- ✓ Отлични преподаватели, ментори и студентска общност



Софтуни предоставя съвременно иновативно образование за програмиране, ИТ и дигитални умения на хиляди млади хора. Програмата "**Софтуерен университет**" изгражда истински **професионалисти в света на програмирането**.

Включете се в цялостната програма по софтуерно инженерство на Софтуни, за да усвояте **най-търсените съвременни софтуерни технологии** чрез модерна учебна методика и с много практически задачи и проекти. **Учебният план** е разработен съвместно с ИТ фирмите и изгражда най-търсените от тях умения.

Софтуни работи пряко с компаниите от софтуерната индустрия, съдейтайки на своите студенти за реализацията им като успешни софтуерни инженери.

ПЪТЯТ НА СТУДЕНТА В СОФТУНИ



1 - 1.5 години



1.5 - 2 години



Programming Basics

Кариерен Старт

Дипломиране

70/100 credits

80/100/150 credits

Кандидатствай

softuni.bg/apply

Глава 6.2. Вложени цикли – изпитни задачи

В предходната глава разгледахме **вложените цикли** и как да ги използваме за рисуване на различни **фигури на конзолата**. Научихме се как да отпечатваме фигури с различни размери, измисляйки подходяща логика на конструиране с използване на **единични и вложени for** цикли в комбинация с различни изчисления и програмна логика:

```
for (var r = 1; r <= 5; r++)
{
    Console.WriteLine("*");
    for (var c = 1; c < 5; c++)
        Console.Write(" *");
    Console.WriteLine();
}
```

Запознахме се и с **конструктора new string**, който дава възможност да се печата даден символ определен от нас брой пъти:

```
string printMe = new string('*', 5);
```

Изпитни задачи

Сега нека решим заедно няколко изпитни задачи, за да затвърдим наученото и да развием още алгоритмичното си мислене.

Задача: чертане на крепост

Да се напише програма, която прочита от конзолата **цяло число n** и чертае **крепост** с ширина **2 * n колони** и височина **n реда** като в примерите по-долу. Лявата и дясната колона във вътрешността си са широки **n / 2**.

Примерен вход и изход

| Вход | Изход | Вход | Изход | Вход | Изход |
|------|-----------------------------|------|-----------------------------------|------|-------------------------------------|
| 3 | /^\v/\^\\ _/_/_ | 4 | /^\v\^\v/\^\\ _/_/_/_ | 5 | /^\v__/\^\\ _/_/_/_/_ |
| | | | | | |

Входни данни

Входът е **цяло число n** в интервала [3...1000].

Изходни данни

Да се отпечатат на конзолата n текстови реда, изобразяващи **крепостта**, точно както в примерите.

Насоки и подсказки

От условието на задачата виждаме, че **входните данни** ще се състоят само от един ред, който ще съдържа в себе си едно **цяло число** в интервала [3...1000]. По тази причина ще използваме **променлива от тип int**.

```
var n =
```

След като вече сме декларирали и инициализирали входните данни, трябва да разделим **крепостта** на три части:

- покрив
- тяло
- основа

От примерите можем да разберем, че **покривът** е съставен от **две кули** и **междинна част**. Всяка кула се състои от начало **/**, среда **^** и край ****.



**** е специален символ в езика C# и използвайки само него в метода **Console.WriteLine(...)**, конзолата няма да го разпечата, затова с **\\"/> показваме на конзолата, че искаме да отпечатаме точно този символ, без да се интерпретира като специален (екранираме го, на английски се нарича "character escaping").**

Средата е с размер, равен на $n / 2$, следователно можем да отделим тази стойност в отделна **променлива**. Тя ще пази **големината на средата на кулата**.

```
var colSize = n / 2;
```

Декларирате втора **променлива**, в която ще пазим **стойността** на частта **между двете кули**. Междинната част на покрива е с размер $2 * n - 2 * \text{colSize} - 4$.

```
var midSize =
```

За да отпечатаме на конзолата **покрива**, ще използваме **new string**, която приема два параметъра (**char**, **int**) и съединява даден символ **n** на брой пъти.

```
Console.WriteLine("/{0}\\{1}/{0}\\",
    new string('^', colSize),
    new string('_', midSize));
```

Тялото на крепостта се състои от начало **|**, среда (**празно място**) и край **|**. Средата от празно място е с големина $2 * n - 2$. Броят на **редовете** за стени, можем да определим от дадените ни примери – **n - 3**.

```

for (var row = 1; row <= n - 3; row++)
{
    Console.WriteLine("|{0}|", new string(' ', [REDACTED]));
}

```

За да нарисуваме предпоследния ред, който е част от основата, трябва да отпечатаме начало |, среда (**празно място**)_(**празно място**) и край |. За да направим това, можем да използваме отново вече декларираните от нас променливи **colSize** и **midSize**, защото от примерите виждаме, че са равни на броя _ в покрива.

```

Console.WriteLine("|{0}{1}{0}|",
    new string(' ', colSize + 1),
    new string('_', [REDACTED]));

```

Добавяме към стойността на **празните места + 1**, защото в примерите имаме **едно** празно място повече.

Структурата на **основата на крепостта** е еднаква с тази на **покрива**. Съставена е от **две кули** и **междинна** част. Всяка една **кула** има начало \, среда _ и край /.

```

Console.WriteLine("\\{0}/{1}\\{0}/",
    new string('_', [REDACTED]),
    new string(' ', [REDACTED]));

```

Тестване в Judge системата

Тествайте решението си: <https://judge.softuni.bg/Contests/Practice/Index/513#0>.

Задача: пеперуда

Да се напише програма, която прочита от конзолата **цяло число n** и чертае **пеперуда** с ширина $2 * n - 1$ колонии височина $2 * (n - 2) + 1$ реда като в примерите по-долу. **Лявата и дясната ѝ част** са широки $n - 1$.

Входни данни

Входът е **цяло число n** в интервала [3...1000].

Изходни данни

Да се отпечатат на конзолата $2 * (n - 2) + 1$ текстови реда, изобразяващи **пеперудата**, точно както в примерите.

Примерен вход и изход

| Вход | Изход | Вход | Изход | Вход | Изход |
|------|---------------------|------|---|------|---|
| 3 | *\ /* @ */ * | 5 | ***\ /*** ---\ /--- ***\ /*** @ ***/ *** ---/ \--- ***/ *** | 7 | *****\ /***** -----\ /----- *****\ /***** -----\ /----- *****\ /***** @ *****/ ***** -----/ \----- *****/ ***** -----/ \----- *****/ ***** |
| | | | | | |

Насоки и подсказки

От условието на задачата виждаме, че **входните данни** ще бъдат прочетени само от един ред, който ще съдържа в себе си едно **цяло число** в интервала [3...1000]. По тази причина ще използваме **променлива от тип int**.

```
var n =
```

Можем да разделим фигурата на 3 части – горно крило, тяло и долно крило. За да начертаем горното крило на пеперудата, трябва да го разделим на части – начало *, среда \ / и край *. След разглеждане на примерите можем да кажем, че началото е с големина **n - 2**.

```
var halfRowSize =
```

Виждаме също така, че горното крило на пеперудата е с размер **n - 2**, затова можем да направим цикъл, който да се повтаря **halfRowSize** пъти.

```
for (int i = 1; i <= ; i++)
{
}
```

От примерите можем да забележим, че на четен ред имаме начало *, среда \ / и край *, а на нечетен – начало -, среда \ / и край -. Следователно, трябва да направим **if-else** проверка дали е **четен** или **нечетен** редът и съответно да отпечатаме един от двата типа редове.

Можем да имплементираме тази идея например така:

```
for (int i = 1; i <= halfRowSize; i++)
{
    if (i % 2 != 0)
    {
```

```

        Console.WriteLine("{0}/ \\"{0}",
            new string('*', halfRowSize));
    }
    else
    {
        Console.WriteLine("  " + " * ".PadRight(halfRowSize));
    }
}

```

За да направим тялото на пеперудата, можем отново да използваме променливата **halfRowSize** и да отпечатаме на конзолата точно един ред. Структурата на тялото е с начало (**празно място**), среда @ и край (**празно място**).

```

Console.WriteLine("{0} @ {0}",
    new string(' ', halfRowSize));

```

Остава да и долното крило, което е еднакво с горното крило.

```

for (int i = 1; i <= halfRowSize; i++)
{
    if (i % 2 == 0)
    {
        Console.WriteLine("  " + " * ".PadRight(halfRowSize));
    }
    else
    {
        Console.WriteLine("  " + " * ".PadRight(halfRowSize));
    }
}

```

Тестване в Judge системата

Тествайте решението си: <https://judge.softuni.bg/Contests/Practice/Index/513#1>.

Задача: знак "Стоп"

Да се напише програма, която прочита от конзолата **цяло число n** и чертае преду-
предителен знак **STOP** с размери като в примерите по-долу.

Входни данни

Входът е **цяло число N** в интервала [3...1000].

Изходни данни

Да се отпечатат на конзолата текстови редове, изобразяващи предупредителния знак STOP, точно както в примерите.

Примерен вход и изход

| Вход | Изход | Вход | Изход |
|------|--|------|---|
| 3 |_____.... ...//_____\\"... ..//_____\\".. .//_____\\". //____STOP!____\\" ____// .____//. ..____//.. | 6 |_____.//_____\\"....//_____\\"....//_____\\".... ...//_____\\".... ..//_____\\".... .//_____\\".... //____STOP!____\\" ____// .____//. ..____//...____//....____//.... |

Насоки и подсказки

От условието на задачата виждаме, че **входните данни** ще бъдат прочетени само от един ред, който ще съдържа в себе си едно **цяло число** в интервала [3...1000]. По тази причина ще използваме **променлива от тип int**.

```
int n = int.Parse(Console.ReadLine());
```

Можем да **разделим** фигурата на **3 части** – горна, средна и долнна. **Горната част** се състои от две подчасти – начален ред и редове, в които знака се разширява. **Началния ред** е съставен от начало ., среда _ и край .. След разглеждане на примерите можем да кажем, че началото е с големина **n + 1** и е добре да отделим тази **стойност** в отделна **променлива**.

```
int dots = ...;
```

Трябва да създадем и втора **променлива**, в която ще пазим **стойността** на **средата на началния ред** и е с големина **2 * n + 1**.

```
var underscores = 2 * n + 1;
```

След като вече сме декларирали и инициализирали двете променливи, можем да отпечатаме на конзолата началния ред.

```
Console.WriteLine("{0}{1}{0}",  
    string.Format("Hello {0}!", "World"));
```

За да начертаем редовете, в които знаци се "разширява", трябва да създадем **ЦИ-КЪЛ**, който да се завърти **n** брой пъти. Структурата на един ред се състои от начало **.**, **//** + среда **_** + **\W** и край **.**. За да можем да използваме отново създадените **променливи**, трябва да намалим **dots** с 1 и **underscopes** с 2, защото ние вече сме отпечатали първия ред, а точките и долните черти в горната част от фигурата на всеки ред **намаляват**.

```
underscopes -= 2;  
dots--;
```

На всяка следваща итерация **началото** и **крайт** намаляват с 1, а **средата** се увеличава с 2.

```
for (int i = 0; i < ; i++)
{
    Console.WriteLine("{0}//{1}\\\\\\{0}",
        new string('.', ),
        new string('_', ));

    underscopes += 2;
    dots--;
}
```

Средната част от фигурата има начало $// + _$, среда **STOP!** и край $_ + \backslash\backslash$. Броят на долните черти е (**underscopes - 5**) / 2.

```
Console.WriteLine("//{0}STOP!{0}\\\\\\",  
    new string(' ',
```

Долната част на фигурата, в която знаци се **смалнява**, можем да направим като отново създадем **цикъл**, който да се завърти **n** брой пъти. Структурата на един ред е начало **.** + **\\"**, среда **_** и край **//** + **..**. Броят на **точките** при първата итерация на цикъла трябва да е 0 и на всяка следваща да се **увеличава** с едно. Следователно приемаме че големината на **точките в долната част от фигурата** е равна на **i**.

```
for (int i = 0; i < n; i++)
{
    Console.WriteLine("{0}\\\\\\{1}//{0}",
        new string('.',   ),
        new string('_',   ));
}
```

За да работи нашата програма правилно, трябва на всяка итерация от **цикъла** да намаляваме броя на _ с 2.

```
for (int i = 0; i < n; i++)
```

```
function stringToBase64(str) {
    var b64 = btoa(str);
    return b64;
}
```

```
underscopes -= .;
```

Тестване в Judge системата

Тествайте решението си: <https://judge.softuni.bg/Contests/Practice/Index/513#2>.

Задача: стрелка

Да се напише програма, която прочита от конзолата **цяло нечетно число n** и чертае **вертикална стрелка** с размери като в примерите по-долу.

Примерен вход и изход

Входни данни

Входът е цяло нечетно число n в интервала [3...79].

Изходни данни

Да се отпечата на конзолата вертикална стрелка, при която "#" (диез) индицира стрелката, а "." – останалото.

Насоки и подсказки

От условието на задачата виждаме, че **входните данни** ще бъдат прочетени само от един ред, който ще съдържа в себе си едно **цяло число** в интервала [3...1000]. По тази причина ще използваме **променлива** от тип **int**.

```
int n = ...;
```

Можем да разделим фигурата на 3 части – горна, средна и долната. **Горната част** се състои от две подчасти – начален ред и тяло на стрелката. От примерите виждаме, че броят на **външните точки** в началния ред и в тялото на стрелката са $(n - 1) / 2$. Тази стойност можем да запишем в **променлива outerDots**.

```
var outerDots = (n - 1) / 2;
```

Броят на **вътрешните точки** в тялото на стрелката е $(n - 2)$. Трябва да създадем **променлива** с име **innerDots**, която ще пази тази стойност.

```
var innerDots = n - 2;
```

От примерите можем да видим структурата на началния ред. Трябва да използваме деклариряните и инициализирани от нас **променливи outerDots** и **n**, за да отпечатаме **началния ред**.

```
Console.WriteLine("{0}{1}{0}",
    new string('.', ...),
    new string('#', ...));
```

За да нарисуваме на конзолата тялото на стрелката, трябва да създадем цикъл, който да се повтори $n - 2$ пъти.

```
for (int i = 0; i < ...; i++)
{
    Console.WriteLine("{0}#{1}#{0}",
        ...,
        ...);
}
```

Средата на фигурата е съставена от начало **#**, среда **.** и край **#**. Броят на **#** е равен на **outerDots** и за това можем да използваме отново същата **променлива**.

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

За да начертаем **долната част на стрелката**, трябва да зададем нови стойности на двете променливи **outerDots** и **innerDots**.

```
outerDots = 1;
innerDots = 2 * n - 5;
```

Тъй като **new string** не може да съедини символ 0 пъти, **цикълът**, който ще направим, трябва да се завърти **n - 2** пъти и отделно да отпечатаме последния ред от фигурата. На всяка итерация **outerDots** се увеличава с 1, а **innerDots** намалява с 2. Ето и примерна имплементация на описаното:

```
for (int i = 0; i < n - 2; i++)
{
    Console.WriteLine("*****",
        "*****", "*",
        );
    outerDots++;
    innerDots -= 2;
}
```

Последният ред от нашата фигура е съставен от начало **.**, среда **#** и край **..**. Броят на **.** е равен на **outerDots**.

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

Тестване в Judge системата

Тествайте решението си: <https://judge.softuni.bg/Contests/Practice/Index/513#3>.

Задача: брадва

Да се напише програма, която прочита **цяло число n** и чертае брадва с размери, показани по-долу. Ширината на брадвата е **5 * N** колони.

Примерен вход и изход

| Вход | Изход | Вход | Изход |
|------|--|------|---|
| 2 | <pre>-----**----- -----*-*- *****-*- -----***-</pre> | 5 | <pre>-----**----- -----*-*- -----*-*- -----*-*- -----*-*- *****-*- *****-*- -----*-*- -----*****-</pre> |

| Вход | Изход |
|------|---|
| 8 | <pre>-----**----- -----*-*- -----*-*- -----*-*- -----*-*- -----*-*- -----*-*- -----*-*- -----*-*- -----*-*- -----*-*- -----*-*- -----*-*- -----*-*- -----*-*- -----*-*- -----*-*- -----*****-</pre> |

Входни данни

Входът е цяло число n в интервала [2..42].

Изходни данни

Да се отпечата на конзолата брадва, точно както е в примерите.

Насоки и подсказки

За решението на задачата е нужно първо да изчислим големината на тиретата от ляво, средните тирета, тиретата от дясно и цялата дължина на фигурата.

```

width = 5 * n;
leftDashes = 3 * n;
middleDashes = 0;
rightDashes = width - leftDashes - middleDashes - 2;

```

След като сме декларирали и инициализирали **променливите**, можем да започнем да изчертаваме фигурата като започнем с **горната част**. От примерите можем да разберем каква е структурата на **първия ред** и да създадем цикъл, който се повтаря **n** брой пъти. На всяка итерация от цикъла **средните тирета** се увеличават с 1, а **тиретата от дясно** се намаляват с 1.

```

for (int i = 0; ; i++)
{
    Console.WriteLine("{0}*{1}*{2}",
        new string(      ),
        new string(      ),
        new string(      ));
}

```

За да можем да използваме отново създадените **променливи** при чертането на дръжката на брадвата, трябва да намалим **средните тирета** с 1, а **тиретата от дясно** да увеличим с 1:

```

middleDashes ;
rightDashes ;

```

Дръжката на брадвата можем да нарисуваме, като завъртим цикъл, който се повтаря **n - 2** пъти. От примерите можем да разберем, каква е нейната структура.

Ето и идея за примерен код как конкретно може да стане това:

```

(int i = 0; ; i++)
{
    Console.WriteLine("      ",
        new string(      ),
        new string(      ),
        new string(      ));
}

```

Долната част на фигурата, трябва да разделим на две подчасти – **глава на брадвата** и **последния ред** от фигурата.

Главата на брадвата ще отпечатаме на конзолата, като направим цикъл, който да се повтаря $n / 2 - 1$ пъти. На всяка итерация тиретата от ляво и тиретата от дясно намаляват с 1, а средните тирета се увеличават с 2.

Ето и идея за примерна имплементация:

```
for ( ... n / 2 - 1 )
{
    Console.WriteLine("{0}*{1}*{2}",
        new string( ... n / 2 - 1 ),
        new string( ... n / 2 - 1 ),
        new string( ... n / 2 - 1 ));

    ... n / 2 - 1
}

}
```

За последния ред от фигурата, можем отново да използваме трите, вече декларирани и инициализирани променливи `leftDashes`, `middleDashes`, `rightDashes`.

```
Console.WriteLine("      ",
    new string( ... n / 2 - 1 ),
    new string( ... n / 2 - 1 ),
    new string( ... n / 2 - 1 ));
```

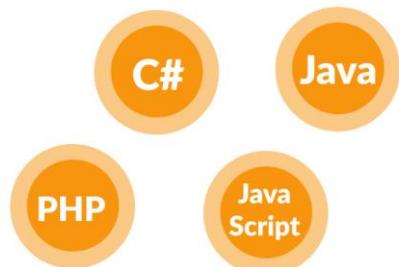
Тестване в Judge системата

Тествайте решението си: <https://judge.softuni.bg/Contests/Practice/Index/513#4>.

Качествено образование,
професия и работа за

Софтуерни инженери

- ✓ Безплатен старт за начинаещи - **присъствено и онлайн**
- ✓ Избор измежду **най-търсените професии** в софтуерната индустрия
- ✓ Съдействие за **кариерен старт**
- ✓ Отлични преподаватели, ментори и студентска общност



Софтуни предоставя съвременно иновативно образование за програмиране, ИТ и дигитални умения на хиляди млади хора. Програмата "**Софтуерен университет**" изгражда истински **професионалисти в света на програмирането**.

Включете се в цялостната програма по софтуерно инженерство на Софтуни, за да усвоите **най-търсените съвременни софтуерни технологии** чрез модерна учебна методика и с много практически задачи и проекти. **Учебният план** е разработен съвместно с ИТ фирмите и изгражда най-търсените от тях умения.

Софтуни работи пряко с компаниите от софтуерната индустрия, съдейтайки на своите студенти за реализацията им като успешни софтуерни инженери.

ПЪТЯТ НА СТУДЕНТА В СОФТУНИ



1 - 1.5 години



1.5 - 2 години



Programming Basics

Кариерен Старт

Дипломиране

70/100 credits

80/100/150 credits

Кандидатствай

softuni.bg/apply

Глава 7.1. По-сложни цикли

След като научихме какво представляват и за какво служат **for циклите**, сега предстои да се запознаем с **други видове цикли**, както и с някои **по-сложни конструкции за цикъл**. Те ще разширят познанията ни и ще ни помогнат в решаването на по-трудни и по-предизвикателни задачи. По-конкретно, ще разгледаме как се ползват следните програмни конструкции:

- цикли **със стъпка**
- **while** цикли
- **do-while** цикли
- безкрайни цикли

В настоящата тема ще разберем и какво представлява операторът **break**, както и как чрез него да прекъснем един цикъл. Също така, използвайки **try-catch** конструкцията, ще се научим да следим за **грешки** по време на изпълнението на програмата ни.

Видео

Гледайте видео-урок по тази глава: <https://youtube.com/watch?v=lovQ8OTnYuQ>.

Цикли със стъпка

В главата "[Повторения \(цикли\)](#)" научихме как работи **for** цикълът и вече знаем кога и с каква цел да го използваме. В тази тема ще обрнем **внимание** на една определена и много важна **част от конструкцията** му, а именно **стъпката**.

Какво представлява стъпката?

Стъпката е тази част от конструкцията на **for** цикъла, която указва с **колко** да се **увеличи** или **намали** стойността на **водещата** му променлива. Тя се декларира последна в скелета на **for** цикъла. При стъпка 10, цикълът би изглеждал по следния начин:

```
int n = int.Parse(Console.ReadLine());
for (var i = 1; i <= n; i+=10)
{
    Console.WriteLine(i);
}
```

Задаване на
стъпка

Най-често стъпката е с **размер 1** и в такъв случай, вместо да пишем **i += 1** или **i -= 1**, можем да използваме операторите **i++** или **i--**. Ако искаме стъпката ни да е **различна от 1**, при увеличение използваме оператора **i += + размера на стъпката**, а при намаляване **i -= + размера на стъпката**.

Следва поредица от примерни задачи, решението на които ще ни помогне да разберем по-добре употребата на **стъпката** във **for** цикъл.

Пример: числата от 1 до N през 3

Да се напише програма, която отпечатва числата от 1 до n със стъпка 3. Например, ако $n = 100$, то резултатът ще е: 1, 4, 7, 10, ..., 94, 97, 100.

Можем да решим задачата чрез следната поредица от действия (алгоритъм):

- Четем числото **n** от входа на конзолата.
- Изпълняваме **for цикъл** от 1 до **n** с размер на стъпката 3.
- В тялото на цикъла отпечатваме стойността на текущата стъпка.

```
int n = int.Parse(Console.ReadLine());
// чрез i+=3 повишаваме стойността на i с размера на стъпката
for (var i = 1; i <= n; i += 3)
{
    Console.WriteLine(i);
}
```

Тестване в Judge системата

Тествайте решението си: <https://judge.softuni.bg/Contests/Practice/Index/514#0>.

Пример: числата от N до 1 в обратен ред

Да се напише програма, която отпечатва числата от n до 1 в обратен ред (със стъпка -1). Например, ако $n = 100$, то резултатът ще е: 100, 99, 98, ..., 3, 2, 1.

Да погледнем нужните действия за да решим задачата:

- Четем числото **n** от входа на конзолата.
- Създаваме **for цикъл** с размер на стъпката **-1**.
 - Старираме цикъла с начална стойност: **int i = n**
 - Обръщаме условието на цикъла: **i >= 1**
 - Дефинираме размера на стъпката: **-1**
- В тялото на цикъла отпечатваме стойността на текущата стъпка **i**.

Ето и **примерно решение** на цялата задача:

```
int n = int.Parse(Console.ReadLine());
// Обърнатото условие: i >= 1
// Отрицателна стъпка: i--
for (int i = n; i >= 1; i--)
{
    Console.WriteLine(i);
}
```

Тестване в Judge системата

Тествайте решението си: <https://judge.softuni.bg/Contests/Practice/Index/514#1>.

Пример: числата от 1 до 2^n с for цикъл

В следващия пример ще разгледаме ползването на обичайната стъпка с размер 1, но с малко по-различна логика в тялото на цикъла.

Да се напише програма, която отпечатва числата от 1 до 2^n (две на степен n). Например, ако $n = 10$, то резултатът ще е 1, 2, 4, 8, 16, 32, 64, 128, 256, 512, 1024.

```
int n = int.Parse(Console.ReadLine());
int num = 1;
for (var i = 0; i <= n; i++)
{
    Console.WriteLine(num);
    num = num * 2;
}
```

Тестване в Judge системата

Тествайте решението си: <https://judge.softuni.bg/Contests/Practice/Index/514#2>.

Пример: четни степени на 2

Да се отпечатат четните степени на 2 до 2^n : $2^0, 2^2, 2^4, 2^8, \dots, 2^n$. Например, ако $n = 10$, то резултатът ще е 1, 4, 16, 64, 256, 1024.

Ето една идея как можем да решим задачата:

- Създаваме променлива **num** за текущото число, на която присвояваме начална стойност 1.
- За стъпка на цикъла задаваме стойност 2.
- В тялото на цикъла: отпечатваме стойността на текущото число и увеличаваме текущото число **num** 4 пъти (според условието на задачата).

Ето го и цялото решение:

```
int n = int.Parse(Console.ReadLine());
int num = 1;
for (var i = 0; i <= n; i += 2)
{
    Console.WriteLine(num);
    num = num * 2 * 2;
}
```

Тестване в Judge системата

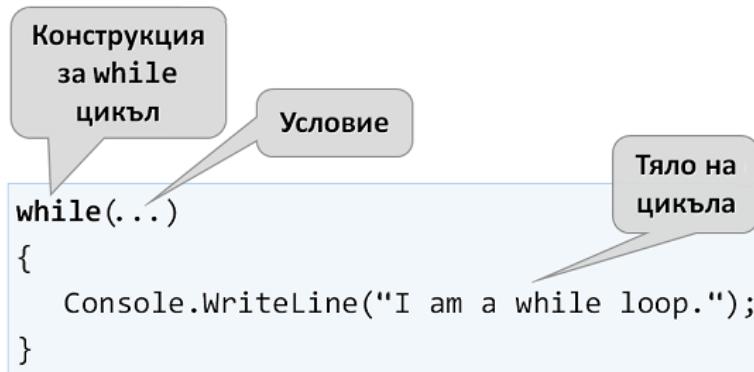
Тествайте решението си: <https://judge.softuni.bg/Contests/Practice/Index/514#3>.

While цикъл

Следващият вид цикли, с които ще се запознаем, се наричат **while** цикли. Специфичното при тях е, че повтарят блок от команди, **докато дадено условие е истина**. Като структура се различават от тази на **for** циклите, даже имат опростен синтаксис.

Какво представлява while цикълът?

В програмирането **while** цикълът се използва, когато искаме да **повтаряме** определена логика, докато **е в сила дадено условие**. Под "условие", разбираме всеки израз, който връща **true** или **false**. Когато **условието стане грешно**, **while** цикълът прекъсва изпълнението си и програмата продължава с изпълнението на **останалия** код след цикъла. Конструкцията за **while** цикъл изглежда по този начин:



Следва поредица от примерни задачи, решението на които ще ни помогне да разберем по-добре употребата на **while** цикъла.

Пример: редица числа $2k+1$

Да се напише програма, която отпечатва всички **числа $\leq n$** от редицата: **1, 3, 7, 15, 31...**, като приемем, че всяко следващо число = **предишно число * 2 + 1**.

Ето как можем да решим задачата:

- Създаваме променлива **num** за текущото число, на която присвояваме начална **стойност 1**.
- За условие на цикъла слагаме **текущото число $\leq n$** .
- В **тялото на цикъла**: отпечатваме стойността на текущото число и увеличаваме текущото число, използвайки формулата от условието на задачата.

Ето и примерна реализация на описаната идея:

```

int n = int.Parse(Console.ReadLine());
int num = 1;
while (num <= n)
{
    Console.WriteLine(num);
    num = 2 * num + 1;
}

```

Тестване в Judge системата

Тествайте решението си: <https://judge.softuni.bg/Contests/Practice/Index/514#4>.

Пример: число в диапазона [1...100]

Да се въведе **цяло** число в диапазона [1...100]. Ако въведеното число е невалидно, да се въведе отново. В случая, за невалидно число ще считаме всяко такова, което **не е** в зададения диапазон.

За да решим задачата, можем да използваме следния алгоритъм:

- Създаваме променлива **num**, на която присвояваме целочислената стойност, получена от входа на конзолата.
- За условие на цикъла слагаме израз, който е **true**, ако числото от входа **не е** в диапазона посочен в условието.
- В **тялото на цикъла**: отпечатваме съобщението "Invalid number!" на конзолата, след което присвояваме нова стойност за **num** от входа на конзолата.
- След като вече сме валидирали въведеното число, извън тялото на цикъла отпечатваме стойността на числото.

Ето и примерна реализация на алгоритъма чрез **while** цикъл:

```

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);

```

Тестване в Judge системата

Тествайте решението си: <https://judge.softuni.bg/Contests/Practice/Index/514#5>.

Най-голям общ делител (НОД)

Преди да продължим към следващата задача, е необходимо да се запознаем с определението за **най-голям общ делител (НОД)**.

Определение за НОД: най-голям общ делител на две **естествени** числа **a** и **b** е най-голямото число, което дели **едновременно** и **a**, и **b** без остатък. Например:

| a | b | НОД | a | b | НОД |
|----|----|-----|-----|----|-----|
| 24 | 16 | 8 | 15 | 9 | 3 |
| 67 | 18 | 1 | 10 | 10 | 10 |
| 12 | 24 | 12 | 100 | 88 | 4 |

Алгоритъм на Евклид

В следващата задача ще използваме един от първите публикувани алгоритми за намиране на НОД – **алгоритъм на Евклид**:

Докато не достигнем остатък 0:

- Делим по-голямото число на по-малкото.
- Вземаме остатъка от делението.

Псевдо-код за алгоритъма на Евклид:

```
while b ≠ 0
    var oldB = b;
    b = a % b;
    a = oldB;
print a;
```

Прочетете повече за алгоритъма на Евклид в Уикипедия: https://en.wikipedia.org/wiki/Euclidean_algorithm.

Пример: най-голям общ делител (НОД)

Да се въведат **цели** числа **a** и **b** и да се намери **НОД(a, b)**.

Ще решим задачата чрез **алгоритъма на Евклид**:

- Създаваме променливи **a** и **b**, на които присвояваме **целочислени** стойности, взети от входа на конзолата.
- За условие на цикъла слагаме израз, който е **true**, ако числото **b** е различно от 0.
- В **тялото на цикъла** следваме указанията от псевдо кода:
 - Създаваме временна променлива, на която присвояваме **текущата** стойност на **b**.

- Присвояваме нова стойност на **b**, която е остатъка от делението на **a** и **b**.
- На променливата **a** присвояваме **предишната** стойност на **b**.
- След като цикълът приключи и сме установили НОД, го отпечатваме на екрана.

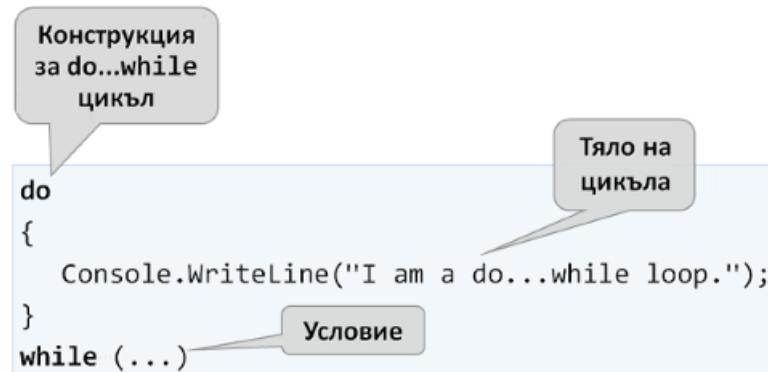
```
var a = int.Parse(Console.ReadLine());
var b = int.Parse(Console.ReadLine());
while (b != 0)
{
    var oldB = b;
    b = a % b;
    a = oldB;
}
Console.WriteLine("GCD = {0}", a);
```

Тестване в Judge системата

Тествайте решението си: <https://judge.softuni.bg/Contests/Practice/Index/514#6>.

Do-while цикъл

Следващият цикъл, с който ще се запознаем, е **do-while**, в превод – “правидокато”. По структура, той наподобява **while**, но има съществена разлика между тях. Тя се състои в това, че **do-while** ще изпълни тялото си **поне веднъж**. Защо се случва това? В конструкцията на **do-while** цикъла, условието винаги се проверява **след** тялото му, което от своя страна гарантира, че при **първото завъртане** на цикъла, кодът ще се **изпълни**, а проверката за край на цикъл ще се прилага върху всяка **следваща** итерация на **do-while**.



Следва обичайната поредица от примерни задачи, чито решения ще ни помогнат да разберем по-добре **do-while** цикъла.

Пример: изчисляване на факториел

За естествено число n да се изчисли $n! = 1 * 2 * 3 * \dots * n$. Например, ако $n = 5$, то резултатът ще бъде: $5! = 1 * 2 * 3 * 4 * 5 = 120$.

Ето как по-конкретно можем да пресметнем факториел:

- Създаваме променливата **n** , на която присвояваме целочислена стойност взета от входа на конзолата.
- Създаваме още една променлива – **$fact$** , чиято начална стойност е 1. Ней ще използваме за изчислението и съхранението на факториела.
- За условие на цикъла ще използваме **$n > 1$** , тъй като всеки път, когато извършим изчисленията в тялото на цикъла, ще намаляваме стойността на **n** с 1.
- В тялото на цикъла:
 - Присвояваме нова стойност на **$fact$** , която е резултат от умножението на текущата стойност на **$fact$** с текущата стойност на **n** .
 - Намаляваме стойността на **n** с **-1**.
- Извън тялото на цикъла отпечатваме крайната стойност на факториела.

```
var n = int.Parse(Console.ReadLine());
var fact = 1;
do
{
    fact = fact * n;
    n--;
}
while (n > 1);
Console.WriteLine(fact);
```

Тестване в Judge системата

Тествайте решението си: <https://judge.softuni.bg/Contests/Practice/Index/514#7>.

Пример: сумиране на цифрите на число

Да се сумират цифрите на цяло **положително** число n . Например, ако $n = 5634$, то резултатът ще бъде: $5 + 6 + 3 + 4 = 18$.

Можем да използваме следната идея, за да решим задачата:

- Създаваме променливата **n** , на която присвояваме стойност, равна на въведеното от потребителя число.
- Създаваме втора променлива – **sum** , чиято начална стойност е 0. Ней ще използваме за изчислението и съхранението на резултата.
- За условие на цикъла ще използваме **$n > 0$** , тъй като след всяко изчисление на резултата в тялото на цикъла, ще премахваме последната цифра от **n** .

- В тялото на цикъла:
 - Присвояваме нова стойност на **sum**, която е резултат от събирането на текущата стойност на **sum** с последната цифра на **n**.
 - Присвояваме нова стойност на **n**, която е резултат от премахването на последната цифра от **n**.
- Извън тялото на цикъла отпечатваме крайната стойност на сумата.

Ето и примерна имплементация:

```
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);
```



Запомнете как се извличат цифри от цяло число:

- **n % 10**: връща последната цифра на числото **n**.
- **n / 10**: изтрива последната цифра на **n**.

Тестване в Judge системата

Тествайте решението си: <https://judge.softuni.bg/Contests/Practice/Index/514#8>.

Безкрайни цикли и операторът **break**

До момента се запознахме с различни видове цикли, като научихме какви конструкции имат те и как се прилагат. Следва да разберем какво е **безкраен цикъл**, кога възниква и как можем да прекъснем изпълнението му чрез оператора **break**.

Безкраен цикъл. Що е то?

Безкраен цикъл наричаме този цикъл, който **повтаря безкрайно** изпълнението на тялото си. При **while** и **do-while** циклите проверката за край е условен израз, който **винаги** връща **true**. Безкраен **for** възниква, когато **липсва** условие за край.

Ето как изглежда **безкраен while** цикъл:

```
while (true)
{
    Console.WriteLine("Infinite loop");
```

А така изглежда **безкраен for цикъл**:

```
for (;;)
{
    Console.WriteLine("Infinite loop");
}
```

Оператор break

Вече знаем, че безкрайният цикъл изпълнява определен код до безкрайност, но какво става, ако желаем в определен момент при дадено условие, да излезем принудително от цикъла? На помощ идва операторът **break**, в превод – **спри, прекъсни**.



Операторът **break** спира изпълнението на цикъла към момента, в който е извикан, и продължава от първия ред след края на цикъла. Това означава, че текущата итерация на цикъла няма да бъде завършена до край и съответно останалата част от кода в тялото на цикъла няма да се изпълни.

Пример: прости числа

В следващата задача се изиска да направим **проверка за просто число**. Преди да продължим към нея, нека си припомним какво са простите числа.

Определение: едно цяло число е **просто**, ако се дели без остатък единствено на себе си и на 1. По дефиниция простите числа са положителни и по-големи от 1. Най-малкото просто число е **2**.

Можем да приемем, че цяло число **n** е просто, ако $n > 1$ и **n** не се дели на число между **2** и **n-1**.

Първите няколко прости числа са: 2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, ...

За разлика от тях, **непростите (композитни) числа** са такива числа, чиято композиция е съставена от произведение на прости числа.

Ето няколко примерни непрости числа:

- $10 = 2 * 5$
- $42 = 2 * 3 * 7$
- $143 = 13 * 11$

Алгоритъм за проверка дали дадено цяло число е **просто**: проверяваме дали $n > 1$ и **n** се дели на **2, 3, ..., n-1** без остатък.

- Ако се раздели на някое от числата, значи е **композитно**.
- Ако не се раздели на никое от числата, значи е **просто**.



Можем да оптимизираме алгоритъма, като вместо проверката да е до $n-1$, да се проверяват делителите до \sqrt{n} . Помислите защо.

Пример: проверка за просто число. Оператор break

Да се провери дали едно число n е просто. Това ще направим като проверим дали n се дели на числата между 2 и \sqrt{n} .

Ето го алгоритъмът за проверка за просто число, разписан постъпково:

- Създаваме променливата **n** , на която присвояваме цяло число въведено от входа на конзолата.
- Създаваме булева променлива **isPrime** с начална стойност **true**. Приемаме, че едно число е просто до доказване на противното.
- Създаваме **for** цикъл, на който като начална стойност за променливата на цикъла задаваме 2, за условие **текущата ѝ стойност $\leq \sqrt{n}$** . Стъпката на цикъла е 1.
- В **тялото на цикъла** проверяваме дали **n** , разделено на **текущата стойност** има остатък. Ако от делението **няма остатък**, то променяме **isPrime** на **false** и излизаме принудително от цикъла чрез оператор **break**.
- В зависимост от стойността на **isPrime** отпечатваме дали числото е просто (**true**) или съответно (**false**).

Ето и примерна имплементация на описания алгоритъм:

```
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");
}
```

Оставаме да добавите проверка дали входното число е по-голямо от 1, защото по дефиниция числа като 0, 1, -1 и -2 не са прости.

Тестване в Judge системата

Тествайте решението си: <https://judge.softuni.bg/Contests/Practice/Index/514#9>.

Пример: оператор **break** в безкраен цикъл

Да се напише програма, която проверява дали едно число **n** е четно, и ако е – да се отпечатва на екрана. За четно считаме число, което се дели на 2 без остатък. При невалидно число да се връща към повторно въвеждане и да се изписва съобщение, което известява, че въведеното число не е четно.

Ето една идея как можем да решим задачата:

- Създаваме променлива **n**, на която присвояваме начална стойност 0.
- Създаваме безкраен **while** цикъл, като за условие ще зададем **true**.
- В тялото на цикъла:
 - Вземаме целочислена стойност **n** от входа на конзолата.
 - Ако **числото n е четно**, излизаме от цикъла чрез **break**.
 - В **противен случай** извеждаме съобщение, което гласи, че **числото не е четно**. Итерациите продължават, докато не се въведе четно число.
- Отпечатваме четното число на екрана.

Ето и примерна имплементация на идеята:

```
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);
```

Забележка: макар кодът по-горе да е коректен, той няма да работи, ако вместо числа потребителят въведе текст, например **"Invalid number"**. Тогава парсването на текста към число ще се счупи и програмата ще покаже **съобщение за грешка (изключение)**. Как да се справим с този проблем и как да прихващаме и обработваме изключения чрез **try-catch** конструкцията ще научим след малко.

Тестване в Judge системата

Тествайте решението си: <https://judge.softuni.bg/Contests/Practice/Index/514#10>.

Вложени цикли и операторът `break`

След като вече научихме какво са **вложените цикли** и как работи операторът **`break`**, е време да разберем как работят двете заедно. За по-добро разбиране, нека стъпка по стъпка да напишем **програма**, която трябва да направи всички възможни комбинации от **двойки числа**. Първото число от комбинацията е нарастващо от 1 до 3, а второто е намаляващо от 3 до 1. Задачата трябва да продължи изпълнението си, докато **`i + j` не** е равно на 2 (т.е. **`i = 1` и `j = 1`**).

Желаният резултат е:

```
C:\WINDOWS\system32\cmd.exe
1 3
1 2
Press any key to continue . . .
```

Ето едно **грешно решение**, което изглежда правилно на пръв поглед:

```
for (int i = 1; i <= 3; i++)
{
    for (int j = 3; j >= 1; j--)
    {
        if (i + j == 2)
        {
            break;
        }
        Console.WriteLine(i + " " + j);
    }
}
```

Ако оставим програмата ни по този начин, резултатът ни ще е следният:

```
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 . . .
```

Защо се получава така? Както виждаме, в резултата липсва "1 1". Когато програмата стига до там, че **i = 1** и **j = 1**, тя влиза в **if** проверката и изпълнява **break** операцията. По този начин се **излиза от вътрешния цикъл**, но след това продължава изпълнението на външния. **i** нараства, програмата влиза във вътрешния цикъл и принтира резултата.



Когато във **вложен цикъл** използваме оператора **break**, той прекъсва изпълнението **само** на вътрешния цикъл.

Какво е **правилното решение**? Един начин за решаването на този проблем е чрез деклариране на **bool** променлива, която следи за това, дали трябва да продължа-ва въртенето на цикъла. При нужда от изход (излизане от всички вложени цикли), се прави **true** променливата и се излиза от вътрешния цикъл с **break**, а при последваща проверка се напуска и външния цикъл. Ето и примерна имплемента-ция на тази идея:

```
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);
        }
    }
}
```

По този начин, когато **i + j = 2**, програмата ще направи променливата **hasToEnd = true** и ще излезе от вътрешния цикъл. При следващото завъртане на външния цикъл, чрез **if** проверката, програмата няма да може да стигне до вътрешния ци-къл и ще прекъсне изпълнението си.

Тестване в Judge системата

Тествайте решението си: <https://judge.softuni.bg/Contests/Practice/Index/514#11>.

Справяне с грешни данни: try-catch

Последното, с което ще се запознаем в тази глава, е как да "улавяме" грешни данни чрез конструкцията **try-catch**.

Какво е try-catch?

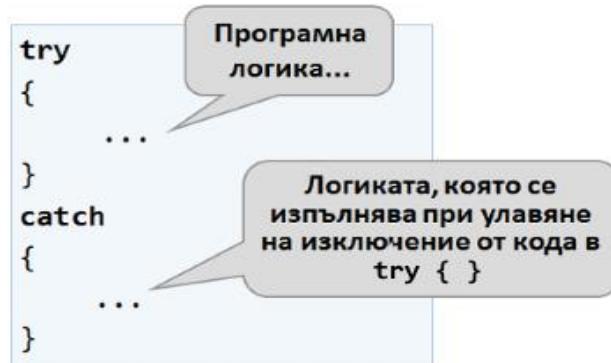
Програмната конструкция **try-catch** служи за прихващане и обработка на изключения (грешки) по време на изпълнението на програмата.

В програмирането **изключенията** представляват уведомление за дадено събитие, което наруши нормалната работа на една програма. Такива изключителни събития **прекъсват изпълнението** на програмата ни и тя търси кой обработи настъпилата ситуация. Ако не намери, изключението се отпечатва на конзолата (програмата "гърми"). Ако намери, **изключението се обработва** и програмата продължава нормалното си изпълнение без да "гърми". След малко ще видим как точно става това.

Когато настъпи изключение, се казва, че изключението е било "хвърлено" (throw exception). От там идва и изразът "улавям изключение" (catch exception).

Конструкция на try-catch

Конструкцията **try-catch** има различни варианти, но за сега ще се запознаем само с най-основния от тях:



В следващата задача ще видим нагледно, как да се справим в ситуация, в която потребителят въвежда вход, различен от число (например **string** вместо **int**), чрез **try-catch**.

Пример: справяне с грешни числа чрез try-catch

Да се напише програма, която проверява дали едно число **n** е четно и ако е, да се отпечатва на екрана. При **невалидно въведено** число да се изписва съобщение, че въведенния вход не е валидно число и въвеждането да продължи отново.

Ето как можем да решим задачата:

- Създаваме безкраен **while** цикъл, като за условие ще зададем **true**.
- В тялото на цикъла:

- Създаваме **try-catch** конструкция.
- В **try** блока пишем програмната логика за четене на потребителския вход, парсването му до число и проверката за четност.
- При **четно число** го отпечатваме и излизаме от цикъла (с **break**). Програмата си е свършила работата и приключва.
- При **нечетно число** отпечатваме съобщение, че се изисква четно число, без да излизаме от цикъла (защото искаме той да се повтори отново).
- Ако **хванем изключение** при изпълнението на **try** блока, изписваме съобщение за невалидно въведено число (цикълът съответно се повтаря, защото не излизаме изрично от него).

Ето и примерна имплементация на описаната идея:

```
while (true)
{
    try
    {
        Console.WriteLine("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.");
    }
}
```

Тестване в Judge системата

Тествайте решението си: <https://judge.softuni.bg/Contests/Practice/Index/514#12>. Сега вече решението трябва да работи винаги: независимо дали въвеждаме цели числа, невалидни числа (например твърде много цифри) или текстове, които не съдържат числа.

Задачи с цикли

В тази глава се запознахме с няколко нови вида цикли, с които могат да се правят повторения с по-сложна програмна логика. Да решим няколко задачи, използвайки новите знания.

Задача: числа на Фиbonacci

Числата на Фиbonacci в математиката образуват редица, която изглежда по следния начин: **1, 1, 2, 3, 5, 8, 13, 21, 34,**

Формулата за образуване на редицата е:

$$\begin{aligned} F_0 &= 1 \\ F_1 &= 1 \\ F_n &= F_{n-1} + F_{n-2} \end{aligned}$$

Примерен вход и изход

| Вход (n) | Изход | Коментар |
|----------|-------|-------------------------|
| 10 | 89 | $F(11) = F(9) + F(8)$ |
| 5 | 8 | $F(5) = F(4) + F(3)$ |
| 20 | 10946 | $F(20) = F(19) + F(18)$ |

| Вход (n) | Изход |
|----------|-------|
| 0 | 1 |
| 1 | 1 |

Да се въведе **цяло** число **n** и да се пресметне **n**-тото число на Фиbonacci.

Насоки и подсказки

Идея за решаване на задачата:

- Създаваме **променлива n**, на която присвояваме целочислена стойност от входа на конзолата.
- Създаваме променливите **f0** и **f1**, на които присвояваме стойност **1**, тъй като така започва редицата.
- Създаваме **for** цикъл с условие **текущата стойност i < n - 1**.
- В **тялото на цикъла**:
 - Създаваме **временна** променлива **fNext**, на която присвояваме следващото число в поредицата на Фиbonacci.
 - На **f0** присвояваме текущата стойност на **f1**.
 - На **f1** присвояваме стойността на временната променлива **fNext**.
- Извън цикъла отпечатваме числото **n**-тото число на Фиbonacci.

Примерна имплементация:

```
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);
```

Тестване в Judge системата

Тествайте решението си: <https://judge.softuni.bg/Contests/Practice/Index/514#13>.

Задача: пирамида от числа

Да се отпечатат числата **1...n** в **пирамида** като в примерите по долу. На първия ред печатаме едно число, на втория ред печатаме две числа, на третия ред печатаме три числа и т.н. докато числата свършат. На последния ред печатаме толкова числа, колкото останат докато стигнем до **n**.

Примерен вход и изход

| Вход | Изход |
|------|------------------------|
| 7 | 1 2 3 4 5 6 7 |

| Вход | Изход |
|------|-----------------|
| 5 | 1 2 3 4 5 |

| Вход | Изход |
|------|-------------------------------|
| 10 | 1 2 3 4 5 6 7 8 9 10 |

Насоки и подсказки

Можем да решим задачата с **два вложени цикъла** (по редове и колони) с печтане в тях и излизане при достигане на последното число. Ето идеята, разписана по-подробно:

- Създаваме променлива **n**, на която присвояваме целочислена стойност от входа на конзолата.
- Създаваме променлива **num** с начална стойност 1. Тя ще пази броя на отпечатаните числа. При всяка итерация ще я **увеличаваме** с 1 и ще я принтираме.
- Създаваме **външен for** цикъл, който ще отговаря за **редовете** в таблицата. Наименуваме променливата на цикъла **row** и ѝ задаваме начална стойност 0. За условие слагаме **row < n**. Размерът на стъпката е 1.

- В тялото на цикъла създаваме **вътрешен for** цикъл, който ще отговаря за **колоните** в таблицата. Наименуваме променливата на цикъла **col** и ѝ задаваме начална стойност 0. За условие слагаме **col < row** (**row** = брой цифри на ред). Размерът на стъпката е 1.
- В тялото на вложния цикъл:
 - Проверяваме дали **col > 1**, ако да – принтираме разстояние. Ако не направим тази проверка, а директно принтираме разстоянието, ще имаме ненужно такова в началото на всеки ред.
 - Отпечатваме числото **num** в текущата клетка на таблицата и го **увеличаваме с 1**.
 - Правим проверка за **num > n**. Ако **num** е по-голямо от **n**, прекъсваме въртенето на **вътрешния цикъл**.
- Отпечатваме **празен ред**, за да преминем на следващия.
- Отново проверяваме дали **num > n**. Ако е по-голямо, прекъсваме изпълнението на **програмата ни** чрез **break**.

Ето и примерна имплементация:

```
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++;
        if (num > n)
        {
            break;
        }
    }
    Console.WriteLine();
    if (num > n)
    {
```

```

        break;
    }
}

```

Тестване в Judge системата

Тествайте решението си: <https://judge.softuni.bg/Contests/Practice/Index/514#14>.

Задача: таблица с числа

Да се отпечатат числата 1...n в таблица като в примерите по-долу.

Примерен вход и изход

| Вход | Изход | Вход | Изход |
|------|-------------------------|------|--|
| 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 |

Насоки и подсказки

Можем да решим задачата с **два вложени цикъла** и малко изчисления в тях:

```

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();
}

```

- Четем от конзолата размера на таблицата в целочислена променлива **n**.
- Създаваме **for** цикъл, който ще отговаря за редовете в таблицата. Наименуваме променливата на цикъла **row** и ѝ задаваме начална **стойност 0**. За условие слагаме **row < n**. Размерът на стъпката е 1.

- В тялото на цикъла създаваме вложен **for** цикъл, който ще отговаря за колоните в таблицата. Наименуваме променливата на цикъла **col** и ѝ задаваме начална стойност 0. За условие слагаме **col < n**. Размерът на стъпката е 1.
- В тялото на вложения цикъл:
 - Създаваме променлива **num**, на която присвояваме резултата от **текущият ред + текущата колона + 1** (+1, тъй като започваме броенето от 0).
 - Правим проверка за **num > n**. Ако **num** е по-голямо от **n**, присвояваме нова стойност на **num** равна на **два пъти n - текущата стойност за num**. Това правим за да не превишаваме **n** в никоя от клетките на таблицата.
 - Отпечатваме числото от текущата клетка на таблицата.
- Отпечатваме **празен ред** във външния цикъл, за минаване на нов ред.

Тестване в Judge системата

Тествайте решението си: <https://judge.softuni.bg/Contests/Practice/Index/514#15>.

Какво научихме от тази глава?

Можем да използваме **for** цикли със стъпка:

```
for (var i = 1; i <= n; i+=3)
{
    Console.WriteLine(i);
}
```

Циклите **while / do-while** се повтарят докато е в сила дадено **условие**:

```
int num = 1;
while (num <= n)
{
    Console.WriteLine(num++);
}
```

Ако се наложи да прекъснем изпълнението на цикъл, го правим с оператора **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);
```

Вече знаем как да хващаме **грешки** по време на изпълнението на програмата ни:

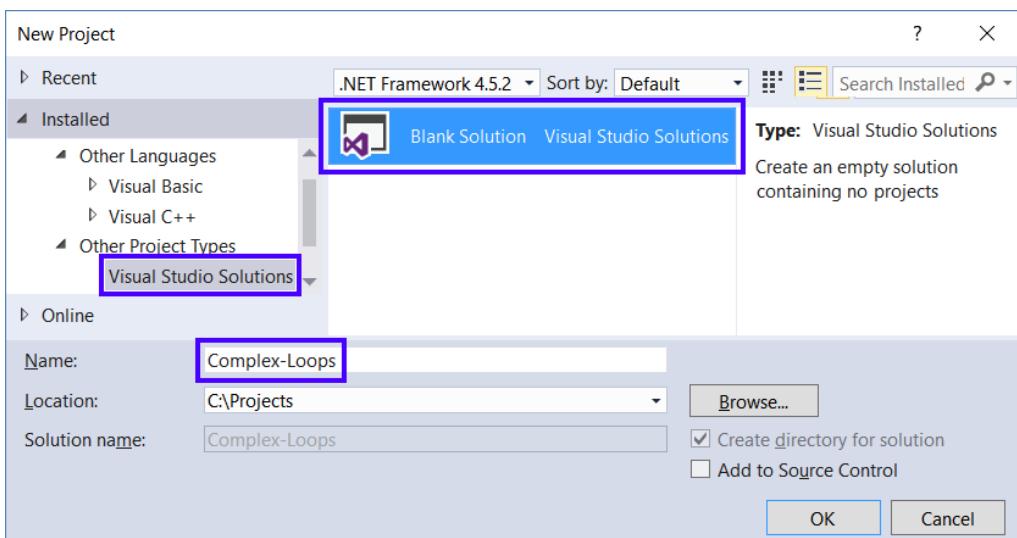
```
try
{
    n = int.Parse(Console.ReadLine());
}
catch
{
    Console.WriteLine("Invalid number.");
}
// Ако int.Parse(...) грямне, ще се изпълни catch { ... } блокът
```

Упражнения: уеб приложения с по-сложни цикли

Сега вече знаем как да повтаряме група действия, използвайки **цикли**. Нека направим нещо интересно: да си направим **уеб базирана игра**. Да, истинска игра, с графика, с гейм логика. Да се позабавляваме. Ще бъде сложно, но ако не разберете нещо как точно работи, няма проблем. Сега още навлизаме в програмирането. Има време, ще напреднете с технологиите. Засега следвайте стъпките.

Празно Visual Studio решение (Blank Solution)

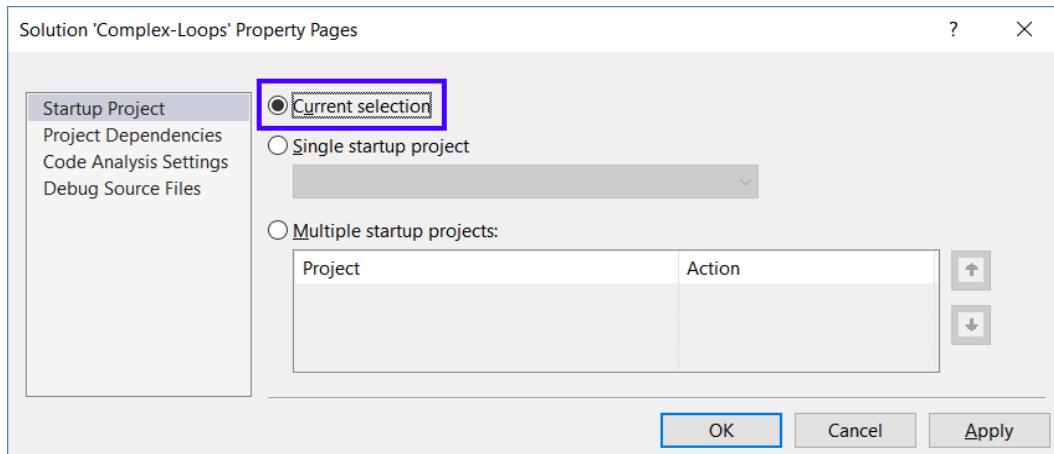
Създайте празно решение (Blank Solution) във Visual Studio, за да организирате кода от задачите за упражнение. Целта на този **blank solution** е да съдържа **по един проект за всяка задача** от упражненията.



Възможно е **.NET версията** при вас да е по-нова. На картинката ползваме **.NET Framework 4.5.2**, но с времето излизат нови версии, така че може да изберете и по-нова или по-стара .NET Framework версия.

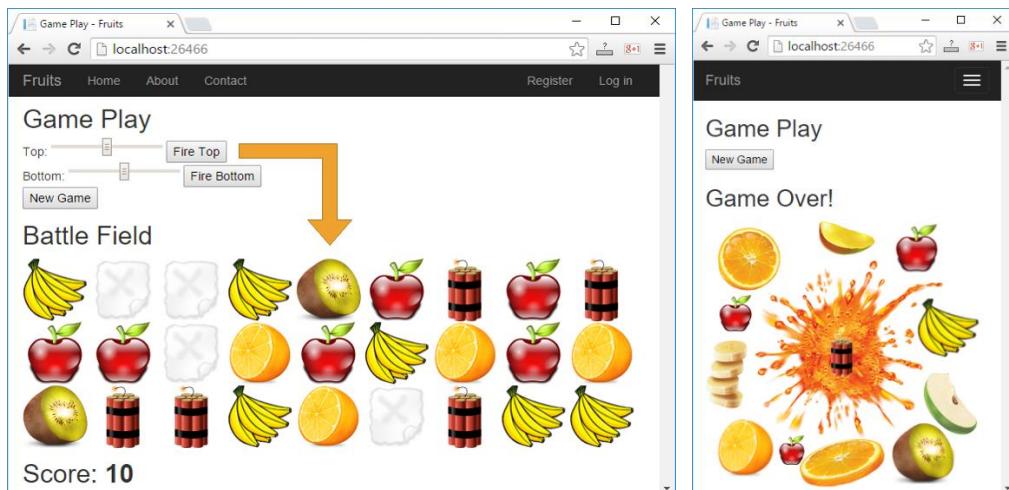
Ако ползвате **Linux** или **Mac OS X**, вместо Visual Studio ще трябва да изберете друга среда за разработка (например **Raider**), а .NET версията ще бъде .NET Core. Някои от стъпките по-долу може да не са същите в Linux среда.

Задайте във Visual Studio да се стартира по подразбиране текущия проект (не първият в решението). Кликнете с десен бутон на мишката върху Solution 'Complex-Loops' -> [Set StartUp Projects...] -> [Current selection].



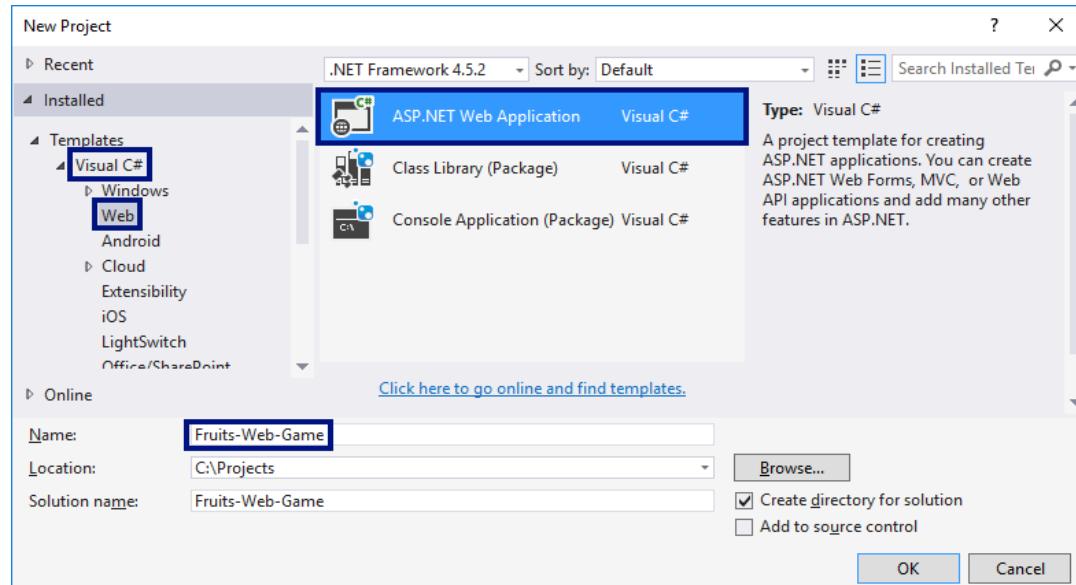
Задача: уеб игра „Обстреляй плодовете!“

Условие: да се разработи **ASP.NET MVC** уеб приложение – игра, в която играчът стреля по плодове, подредени в таблица. Успешно уцелените плодове изчезват, а играчът получава точки за всеки уцелен плод. При уцелване на **динамит**, плодовете се взривяват и играта свършва (както във Fruit Ninja). Стрелбата се извършва по колони, отгоре надолу или отдолу нагоре, а местоположението на удара (колоната под обстрел) се задава чрез скролер (scroll bar). Заради неточността на скролера, играчът не е съвсем сигурен по коя колона ще стреля. Така при всеки изстрел има шанс да не улучи и това прави играта по-интересна (подобно на прашката в Angry Birds). Играча ни трябва да изглежда по този начин:

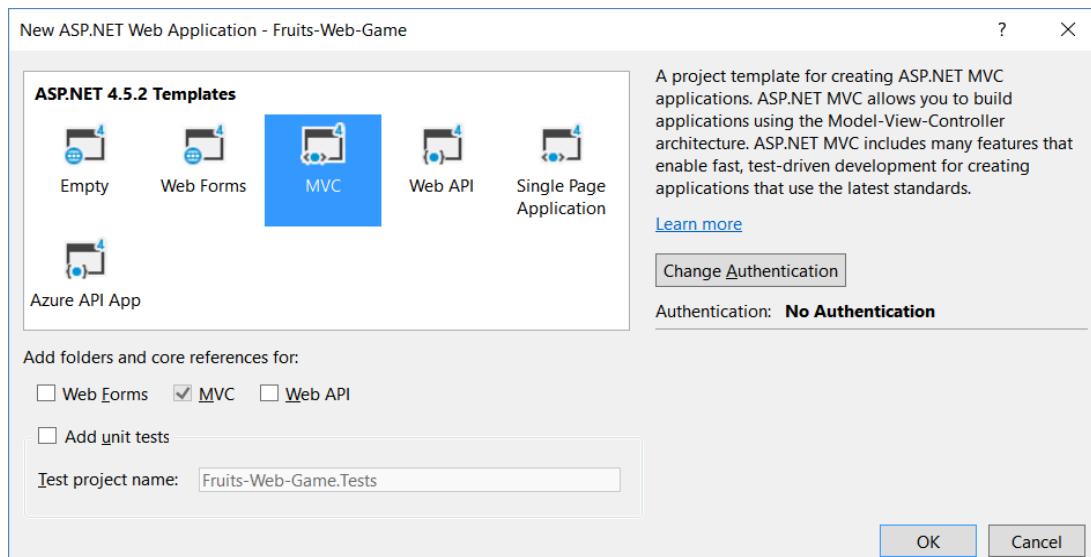


Имплементация на уеб приложението “Обстреляй плодовете!”

Във Visual Studio създаваме ново ASP.NET MVC уеб приложение с език C#. Добавяме нов проект от [Solution Explorer] → [Add] → [New Project...]. Задаваме смислено име, например “Fruits-Web-Game”:



След това избираме тип на уеб приложението "MVC":



Сега създаваме контролите за играта. Целта е да добавим scroll bars (скролиращи ленти,) с които играчът се прицелва, и бутон за старт на **нова игра**. Затова трябва да редактираме файла **Views/Home/Index.cshtml**. Изтриваме всичко в него и въвеждаме кода от картинката:

```

@{
    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>

```

Този код създава уеб форма `<form>` със скролер (поле) `position` за задаване на число в интервала [0...100] и бутон [Fire Top] за изпращане на данните от формата към сървъра. Действието, което ще обработи данните, се назова **Home/FireTop**, което означава метод **FireTop** в контролер **Home**, който се намира във файла **HomeController.cs**. Следват още две подобни форми с бутона [Fire Bottom] и [New Game].

Сега трябва да подгответим плодовете за рисуване в изгледа (view). Добавяме следния код в контролера: **Controllers/HomeController.cs**:

```

public class HomeController : Controller
{
    static int rowsCount = 3;
    static int colsCount = 9;
    static string[,] fruits = GenerateRandomFruits();
    static int score = 0;
    static bool gameOver = false;

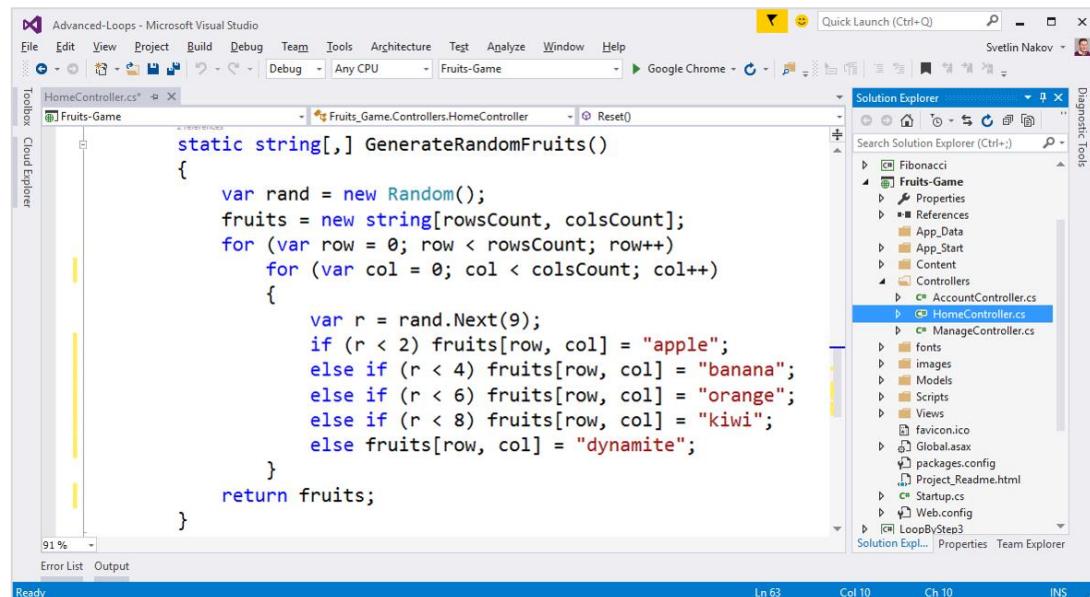
    public ActionResult Index()
    {
        ViewBag.rowsCount = rowsCount;
        ViewBag.colsCount = colsCount;
        ViewBag.fruits = fruits;
        ViewBag.score = score;
        ViewBag.gameOver = gameOver;
        return View();
    }
}

```

Горният код дефинира полета за **брой редове**, **брой колони**, за **таблицата с плодовете** (игралното поле), за натрупаните от играча **точки** и информация дали играта е активна или е **свършила** (поле **gameOver**). Игралното поле е с размери 9 колони на 3 реда и съдържа за всяко поле текст какво има в него: **apple**, **banana**, **orange**, **kiwi**, **empty** или **dynamite**. Главното действие **Index()** подготвя игралното поле за чертане като записва във **ViewBag** структурата елементите на играта и извика изгледа, който ги чертае в страницата на играта в уеб браузъра като HTML.

Трябва да генерираме случаини плодове. За да направим това, трябва да напишем метод **GenerateRandomFruits()** с кода от картинката по-долу. Този код записва в таблицата (матрицата) **fruits** имена на различни картинки и така изгражда игралното поле. Във всяка клетка от таблицата се записва една от следните стойности: **apple**, **banana**, **orange**, **kiwi**, **empty** или **dynamite**. След това, за да се нарисува съответното изображение в изгледа, към текста от таблицата ще се долепи **.png** и така ще се получи името на файла с картинката, която да се вмъкне в HTML страницата като част от игралното поле. Попълването на игралното поле (9 колони с по 3 реда) става в изгледа **Index.cshtml** с два вложени **for** цикъла (за ред и за колона).

За да се генерират случаини плодове за всяка клетка се генерира **случайно число** между 0 и 8 (вж. класа **Random** в .NET). Ако числото е 0 или 1, се слага **apple**, ако е между 2 и 3, се слага **banana** и т.н. Ако числото е 8, се поставя **dynamite**. Така плодовете се появяват 2 пъти по-често отколкото динамита. Ето и кода:



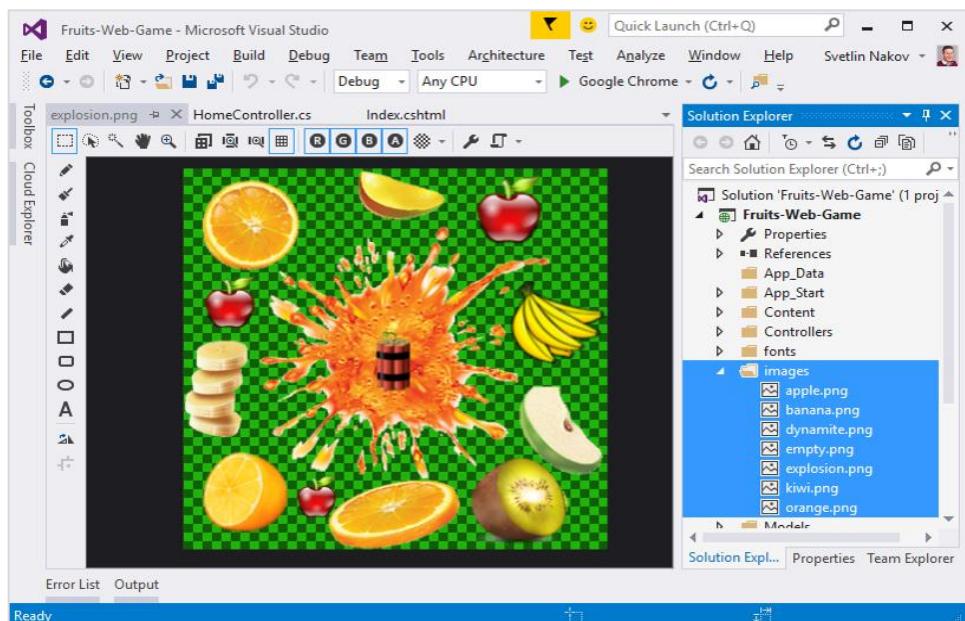
```

static string[,] GenerateRandomFruits()
{
    var rand = new Random();
    fruits = new string[rowsCount, colsCount];
    for (var row = 0; row < rowsCount; row++)
        for (var col = 0; col < colsCount; col++)
    {
        var r = rand.Next(9);
        if (r < 2) fruits[row, col] = "apple";
        else if (r < 4) fruits[row, col] = "banana";
        else if (r < 6) fruits[row, col] = "orange";
        else if (r < 8) fruits[row, col] = "kiwi";
        else fruits[row, col] = "dynamite";
    }
    return fruits;
}

```

Добавяме картинките за играта. От [Solution Explorer] създаваме папка "**images**" в коренната директория на проекта. Използваме менюто **Add** → **New Folder**. Сега добавяме **картинките** за играта (те са част от файловете със заданието за този проект и могат да бъдат свалени от <https://github.com/SoftUni/Programming-Basics-Book-CSharp-BG/tree/master/assets/chapter-7-assets>). Копираме ги от

Windows Explorer и ги поставяме в папката "images" в [Solution Explorer] във Visual Studio с copy / paste.



Чертане на плодовете в **Index.cshtml**: за да начертаем игралното поле с подовете, трябва да завъртим **два вложени цикъла** (за редовете и за колоните). Всеки ред се състои от 9 на брой картички, всяка от които съдържа **apple**, **banana** или друг плод, или празно **empty**, или динамит **dynamite**. Картинките се чертаят като се отпечатва HTML таг за вмъкване на картичка от вида на ``. Девет картички се подреждат една след друга на всеки от редовете, а след тях се преминава на нов ред с `
`. Това се повтаря три пъти за трите реда. Накрая се отпечатват точките на играча. Ето как изглежда **кодът** за чертане на игралното поле и точките:

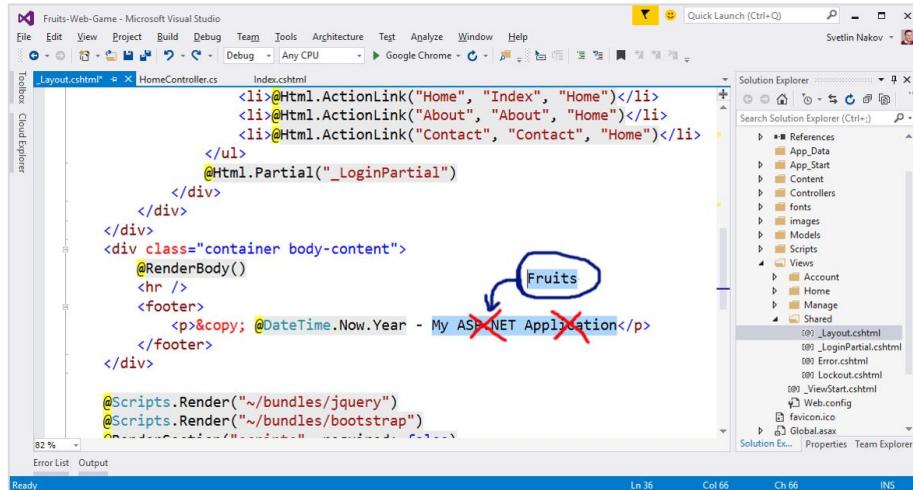
```

<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>

```

Вижте жълтите символи @ – те служат за превключване между езика C# и езика HTML и идват от Razor синтаксиса за рисуване на динамични уеб страници.

Следва да нагласим текстовете във файла `/Views/Shared/_Layout.cshtml`. Заменяме “**My ASP.NET Application**” с по-подходящ текст, напр. “**Fruits**”:



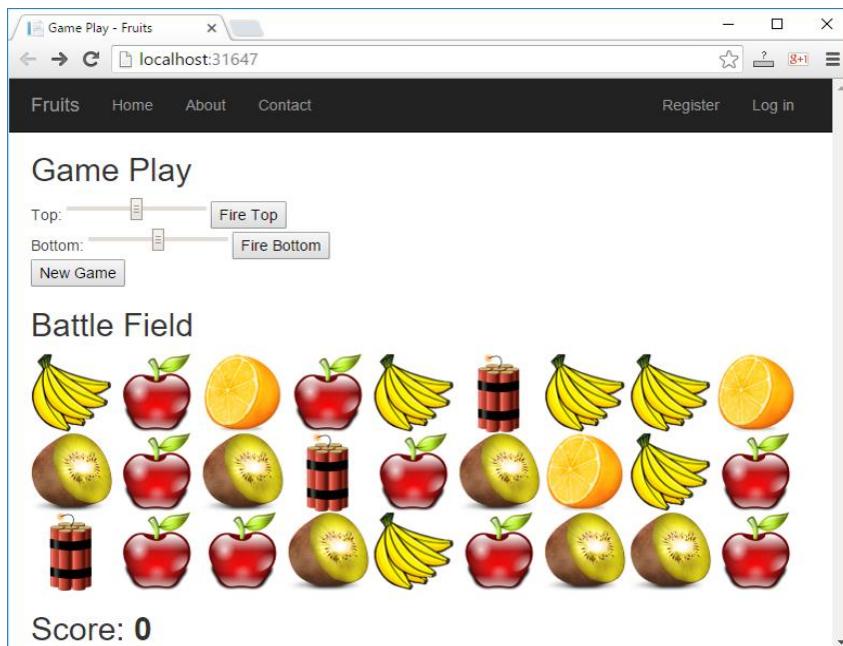
```

<ul>
    <li><a href="#">Home</a></li>
    <li><a href="#">About</a></li>
    <li><a href="#">Contact</a></li>
</ul>
@Html.Partial("_LoginPartial")
</div>
<div class="container body-content">
    @RenderBody()
    <hr />
    <footer>
        <p>&copy; @DateTime.Now.Year - My ASP.NET Application</p>
    </footer>
</div>

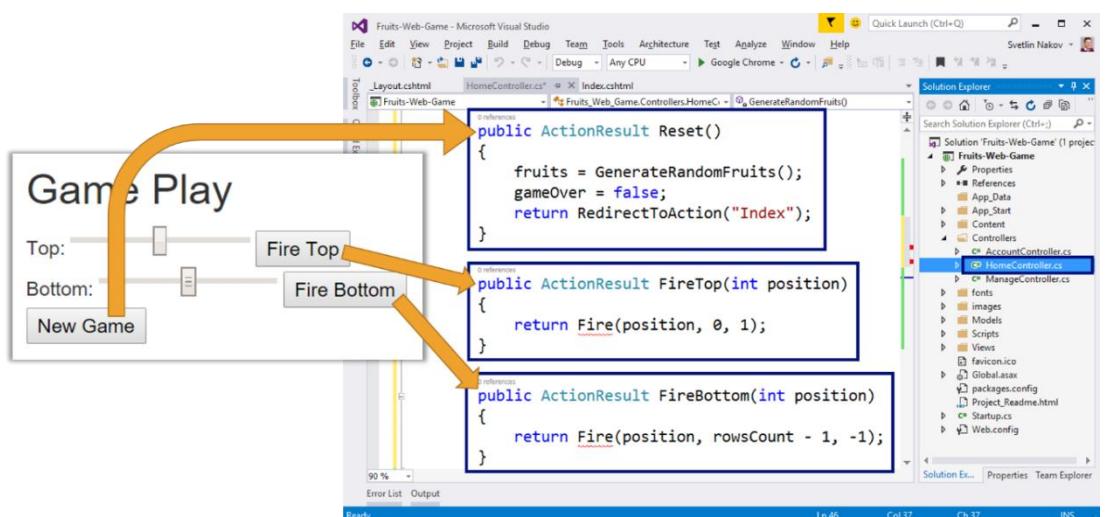
@Scripts.Render("~/bundles/jquery")
@Scripts.Render("~/bundles/bootstrap")

```

Стартираме проекта с [Ctrl+F5] и му се порадвайте. Очаква се да бъде генерирано случайно игрово поле с плодове с размери 9 на 3 и да се визуализира в уеб страницата чрез поредица картинки:



Сега играта е донякъде направена: игралното поле се генерира случайно и се визуализира успешно (ако не сте допуснали грешка някъде). Остава да се реализира същината на играта: **стрелянето по плодовете**. За целта добавяме действията [New Game] и [Fire Top] / [Fire Bottom] в контролера **HomeController.cs**:



Горният код дефинира три действия:

- **Reset()** – стартира нова игра, като генерира ново случайно игрално поле с плодове и експлозиви, нулира точките на играта и прави играта валидна (`gameOver = false`). Това действие е доста просто и може да се тества веднага с [Ctrl+F5], преди да се напишат другите.
- **FireTop(position)** – стреля по ред 0 на позиция `position` (число от 0 до 100). Извиква се стреляне в посока **надолу** (+1) от ред 0 (най-горния). Самото стреляне е по-сложно като логика и ще бъде разгледано след малко.
- **FireBottom(position)** – стреля по ред 2 на позиция `position` (число от 0 до 100). Извиква се стреляне в посока **нагоре** (-1) от ред 2 (най-долния).

Имплементираме "стрелянето" – метода **Fire(position, startRow, step)**:

The screenshot shows the 'HomeController.cs' file in the Visual Studio editor. The 'Fire' method is implemented as follows:

```

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");
}

```

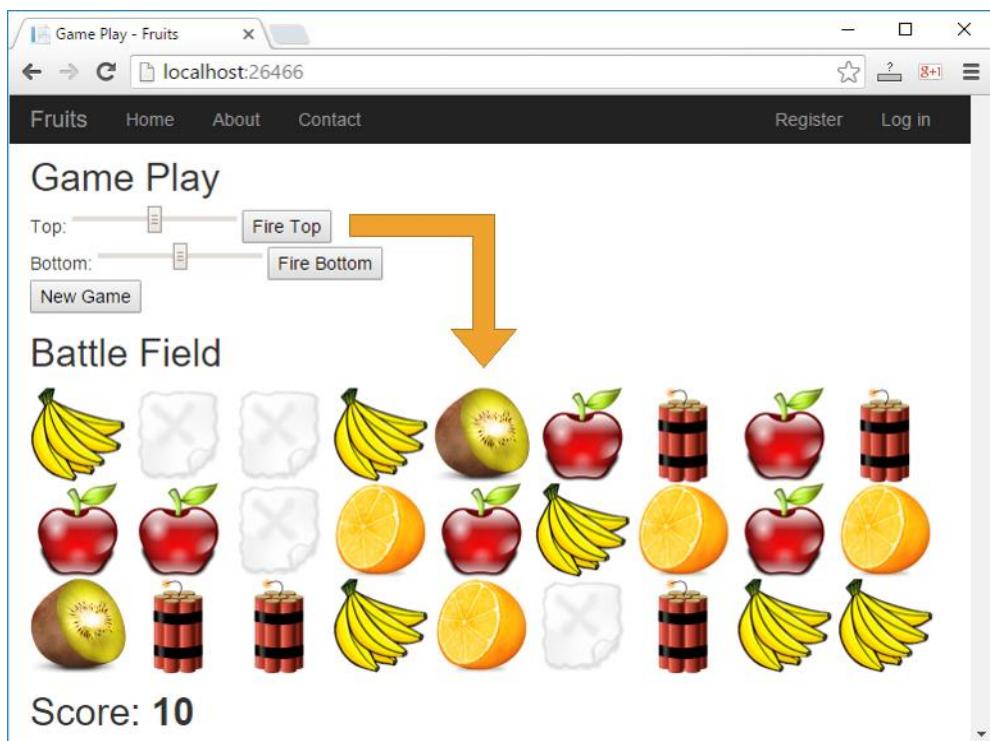
Стрелянето работи по следния начин: първо се изчислява номера на колоната **col**, към която играчът се е прицелил. Входното число от скролера (между 0 и 100) се намалява до число между 0 и 8 (за всяка от 9-те колони). Номерът на реда **row** е или 0 (ако изстрелът е отгоре) или броят редове минус едно (ако изстрелът е отдолу). Съответно посоката на стрелба (стъпката) е **1** (надолу) или **-1** (нагоре).

За да се намери къде изстрелът поразява плод или динамит, се преминава в цикъл през всички клетки от прицелената колона и от първия до последния атакуван ред. Ако се срещне плод, той изчезва (замества се с **empty**) и се дават точки на играча. Ако се срещне **dynamite**, играта се отбелязва като свършила.

Оставаме на по-запалените да имплементират по-сложно поведение, например да се дават различни точки при уцелване на различен плод, да се реализира анимация с експлозия (това не е твърде лесно), да се взимат точки при излишно стреляне в празна колона и подобни.

Тестваме какво работи до момента като стартираме с [Ctrl+F5]:

- **Нова игра** → бутоњът за нова игра трябва да генерира ново игрално поле със случајно разположени плодове и експлозиви и да нулира точките на играча.
- **Стреляне отгоре** → стрелянето отгоре трябва да премахва най-горният плод в уцелената колона или да предизвика край на играта при динамит. Всъщност при край на играта все още нищо няма да се случва, защото в изгледа този случай още не се разглежда.
- **Стреляне отдолу** → стрелянето отдолу трябва да премахва най-долния плод в уцелената колона или да прекратява играта при уцелване на динамит.



За момента при "Край на играта" нищо не се случва. Ако играчът уцели динамит, в контролера се отбелязва, че играта е свършила (**gameOver = true**), но този факт не се визуализира по никакъв начин. За да заработи приключването на играта, е необходимо да добавим няколко проверки в изгледа:

```

@{
    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>

```

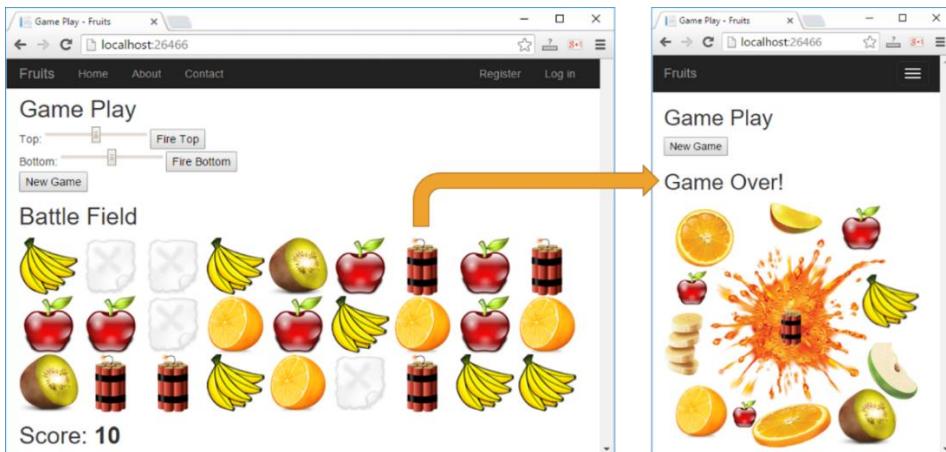
```

@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>
}

```

Кодът по-горе проверява дали е свършила играта и показва съответно контролите за стреляне и игралното поле (при активна игра) или картичка с експлодирали плодове при край на играта.

След промяната в кода на изгледа стартираме играта отново с [Ctrl+F5] и я **тестваме**, за да видим какво се случва при край на играта:



Този път при уцелване на динамит, трябва да се появи дясната картичка и да се позволява единствено действието "нова игра" (бутона [New Game]).

Сложно ли беше? Успяхте ли да направите играта? Ако не сте успели, спокойно, това е сравнително сложен проект, който включва голяма доза не изучавана материя. Ако искате уеб игричката да ви тръгне в ръцете, **гледайте видеото** в началото на тази глава и следвайте стъпките от него. Там приложението е направено на живо с много обяснения. Или питайте за конкретни проблеми във **форума на СофтУни**: <https://softuni.bg/forum>.

Глава 7.2. По-сложни цикли – изпитни задачи

Вече научихме как може да изпълним даден блок от команди повече от веднъж използвайки **for** цикъл. В предходната глава разглеждахме още няколко **циклични конструкции**, които биха ни помогнали при решаването на по-сложни проблеми, а именно:

- цикли със стъпка
- вложени цикли
- **while** цикли
- **do-while** цикли
- безкрайни цикли и излизане от цикъл (**break** оператор)
- конструкцията **try-catch**

Изпитни задачи

Нека затвърдим знанията си като решим няколко по-сложни задачи с цикли, давани на приемни изпити.

Нека затвърдим знанията си като решим няколко по-сложни задачи с цикли, давани на приемни изпити.

Задача: генератор за тъпи пароли

Да се напише програма, която въвежда две цели числа **n** и **l** и генерира по азучен ред всички възможни "тъпи" пароли¹, които се състоят от следните 5 символа:

- Символ 1: цифра от 1 до n.
- Символ 2: цифра от 1 до n.
- Символ 3: малка буква измежду първите l букви на латинската азбука.
- Символ 4: малка буква измежду първите l букви на латинската азбука.
- Символ 5: цифра от 1 до n, по-голяма от първите 2 цифри.

Входни данни

Входът се чете от конзолата и се състои от **две цели числа: n и l** в интервала [1...9], по едно на ред.

Изходни данни

На конзолата трябва да се отпечатат **всички "тъпи"** пароли по азучен ред, разделени с **интервал**.

Примерен вход и изход

| Вход | Изход | Вход | Изход |
|--------|---|--------|----------------------------------|
| 2 4 | 11aa2 11ab2 11ac2 11ad2 11ba2 11bb2 11bc2 11bd2 11ca2 11cb2 11cc2 11cd2 11da2 11db2 11dc2 11dd2 | 3 1 | 11aa2 11aa3 12aa3 21aa3 22aa3 |

| Вход | Изход | Вход | Изход |
|--------|--|--------|---|
| 4 2 | 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 | 3 2 | 11aa2 11aa3 11ab2 11ab3 11ba2 11ba3 11bb2 11bb3 12aa3 12ab3 12ba3 12bb3 21aa3 21ab3 21ba3 21bb3 22aa3 22ab3 22ba3 22bb3 |

Насоки и подсказки

Решението на задачата можем да разделим мислено на **три** части:

- **Прочитане на входните данни** – в настоящата задача това включва прочитането на две числа **n** и **l**, всяко на отделен ред.
- **Обработка на входните данни** – използване на вложени цикли за преминаване през всеки възможен символ за всеки от петте символа на паролата.
- **Извеждане на резултат** – отпечатване на всяка "тъпа" парола, която отговаря на условията.

Прочитане и обработка на входните данни

За прочитане на **входните** данни ще декларираме две променливи от целочислен тип **int: n** и **l**.

```
var n = int.Parse(Console.ReadLine());
var l = int.Parse(Console.ReadLine());
```

Нека декларираме и инициализираме **променливите**, които ще **съхраняват символите** на паролата: за **цифровите** символи – от тип **int** – **d1, d2, d3**, а за **буквените** – от тип **char** – **11, 12**. За улеснение ще пропуснем изричното уточняване на типа като го заместим с ключовата дума **var**.

Извеждане на резултат

Необходимо е да вложим пет **for** цикъла един в друг, по един за всяка променлива. За да гарантираме условието последната цифра **d3** да бъде **по-голяма** от първите две, ще използваме вградената функция **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.WriteLine("{0}{1}{2}{3}{4} ",
                        d1, d2, l1, l2, d3);
                }
                Console.WriteLine();
            }
        }
    }
}
```

Знаете ли, че...?

- Можем да дефинираме **for** цикъл с променлива от тип **char**:

```
for (char ch = 'a'; ch < 'z'; ch++)
```

- Можем да прочетем променлива от тип **char** от конзолата със следната конструкция:

```
char ch = (char)Console.Read();
```

- Можем да обърнем главен символ към малък, използвайки вградена функция в C#:

```
Char.ToLower(ch);
```

- При прочит на символи от конзолата, директно можем да преобразуваме главни към малки букви, **обединявайки горните два реда**:

```
char ch = Char.ToLower((char)Console.Read());
```

Тестване в Judge системата

Тествайте решението си: <https://judge.softuni.bg/Contests/Practice/Index/515#0>

Задача: магически числа

Да се напише програма, която въвежда едно цяло **магическо** число и изкарва всички възможни **6-цифрени** числа, за които произведението на техните цифри е равно на магическото число.

Пример: "Магическо число" → 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 → $2 * 1 * 1 * 1 * 1 * 1 = 2$

Входни данни

Входът се чете от конзолата и се състои от **цяло число** в интервала [1...600000].

Изходни данни

Да се отпечатат на конзолата **всички магически числа**, разделени с **интервал**.

Примерен вход и изход

| Вход | Изход | Вход | Изход | Вход | Изход | |
|------|--|------|--|------|----------------------------|--------|
| 2 | 111112 111121 111211 112111 121111 211111 | 8 | 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 | | 5 3 1 4 4 1 | 999999 |

Насоки и подсказки

Решението на задачата за магическите числа следва **същата** концепция (отново трябва да генерираме всички комбинации за п елемента). Следвайки тези стъпки, опитайте да решите задачата сами.

- Декларирайте и инициализирайте **променлива** от целочислен тип **int** и прочетете **входа** от конзолата.
- Вложете **шест for цикъла** един в друг, по един за всяка цифра на търсените 6-цифрени числа.
- В последния цикъл, чрез **if** конструкция проверете дали **произведенето** на шестте цифри е **равно** на **магическото** число.

```

int magic = int.Parse(Console.ReadLine());
for (int d1 = 0; d1 <= 9; d1++)
{
    for (int d2 = 0; d2 <= 9; d2++)
    {
        for (int d3 = 0; d3 <= 9; d3++)
        {
            for (int d4 = 0; d4 <= 9; d4++)
            {
                for (int d5 = 0; d5 <= 9; d5++)
                {
                    if (d1 * d2 * d3 * d4 * d5 * d6)
                    {
                        Console.WriteLine("The result is " + magic);
                    }
                }
            }
        }
    }
}
}

```

В предходната глава разгледахме и други циклични конструкции. Нека разгледаме примерно решение на същата задача, в което използваме цикъла **while**. Първо трябва да запишем **входното магическо число** в подходяща променлива. След това ще инициализираме 6 променливи – по една за всяка от шестте цифри на търсените като **резултат** числа.

```

int magic = int.Parse(Console.ReadLine());
int d1 = 0;
int d2;

```

```
int d3;
int d4;
int d5;
int d6;
```

След това ще започнем да разписваме **while** циклите.

- Ще инициализираме **първата цифра**: **d1 = 0**.
- Ще зададем **условие за всеки цикъл**: цифрата да бъде по-малка или равна на 9.
- В **началото** на всеки цикъл задаваме стойност на **следващата цифра**, в случая: **d2 = 0**. При вложените **for** цикли инициализираме променливите във вътрешните цикли при всяко увеличение на външните. Искаме да постигнем същото поведение и тук.
- В **края** на всеки цикъл ще **увеличаваме** цифрата с едно: **d++**.
- В **най-вътрешния** цикъл ще направим **проверката** и ако е необходимо, ще принтираме на конзолата.

```
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++;
}

```

Нека премахнем **if** проверката от най-вътрешния цикъл. Сега, нека инициализираме всяка променлива извън циклите и нека изтрием редовете **dx = 0**. След като стартираме програмата, получаваме само 10 резултата. Защо? А ако използвате **do-while**? В случая този цикъл не изглежда подходящ, нали? Помислете защо. Разбира се, можете да решите задачата и с помощта на **безкраен цикъл**.

```

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;
    }
}

```

Както виждаме, един проблем можем да решим с различни видове цикли. Разбира се, за всяка задача има най-подходящ избор. С цел да упражните всички цикъл – опитайте се да решите всяка от следващите задачи с всички изучени цикли.

Тестване в Judge системата

Тествайте решението си: <https://judge.softuni.bg/Contests/Practice/Index/515#1>.

Задача: спиращо число

Напишете програма, която принтира на конзолата всички числа от N до M , които се делят на 2 и на 3 без остатък, в обратен ред. От конзолата ще се чете още едно "спиращо" число S . Ако някое от делящите се на 2 и 3 числа е равно на спиращото число, то не трябва да се принтира и програмата трябва да приключи. В противен случай се принтират всички числа до N , които отговарят на условието.

Вход

От конзолата се четат 3 числа, всяко на отделен ред:

- N – цяло число: $0 \leq N < M$.
- M – цяло число: $N < M \leq 10000$.
- S – цяло число: $N \leq S \leq M$.

Изход

На конзолата се принтират на един ред, разделени с интервал, всички числа, отговарящи на условията.

Примерен вход и изход

| Вход | Изход | Обяснения |
|---------------|---------------|---|
| 1 30 15 | 30 24 18 12 6 | Числата от 30 до 1, които се делят едновременно на 2 и на 3 без остатък са: 30, 24, 18, 12 и 6. Числото 15 не е равно на нито едно, затова редицата продължава. |

| Вход | Изход | Обяснения |
|---------------|-------------|---|
| 1 36 12 | 36 30 24 18 | Числата от 36 до 1, които се делят едновременно на 2 и на 3 без остатък, са: 36, 30, 24, 18, 12 и 6. Числото 12 е равно на спиращото число, затова спираме до 18. |

Насоки и подсказки

Задачата може да се раздели на **четири** логически части:

- **Четене** на входните данни от конзолата.
- **Проверка** на всички числа в дадения интервал, съответно завъртане на цикъл.
- **Проверка** на условията от задачата спрямо всяко едно число от въпросния интервал.

- Разпечатване на числата.

Първата част е тривиална – прочитаме **три** цели числа от конзолата, съответно ще използваме тип **int**.

С **втората** част също сме се сблъсквали – инициализиране на **for** цикъл. Тук има малка **уловка** – в условието е споменато, че числата трябва да се принтират в **обратен ред**. Това означава, че **началната** стойност на променливата **i** ще е **поголямото число**, което от примерите виждаме, че е **M**. Съответно, **крайната** стойност на **i** трябва да е **N**. Фактът, че ще печатаме резултатите в обратен ред и стойностите на **i** ни подсказват, че стъпката ще е **намаляване с 1**.

```
for (int i = m; i >= n; i--)
```

След като сме инициализирали **for** цикъла, идва ред на **третата** част от задачата – **проверка** на условието дали даденото **число се дели на 2 и на 3 без остатък**. Това ще направим с една обикновена **if** проверка, която ще оставим на читателя сам да построи.

Другата **уловка** в тази задача е, че освен горната проверка, трябва да направим **още** една – дали **числото е равно на "спиращото" число**, подадено ни от конзолата на третия ред. За да се стигне до тази проверка, числото, което проверяваме, трябва да премине през горната. По тази причина ще построим още една **if** конструкция, която ще **вложим в предходната**. Ако условието е **вярно**, заданието е да спрем програмата да печата, което в конкретния случай можем да направим с оператор **break**, който ще ни **изведе** от **for** цикъла.

Съответно, ако **условието** на проверката дали числото съвпада със "спиращото" число върне резултат **false**, по задание нашата програма трябва да **продължи да печата**. Това всъщност покрива и **четвъртата и последна** част от нашата програма.

Тестване в Judge системата

Тествайте решението си: <https://judge.softuni.bg/Contests/Practice/Index/515#2>.

Задача: специални числа

Да се напише програма, която **въвежда** едно **цяло** число **N** и генерира всички възможни "специални" числа от **1111** до **9999**. За да бъде "специално" едно число, то трябва да отговаря на **следното условие**:

- **N** да се дели на всяка една от неговите цифри без остатък.

Пример: при **N = 16, 2418** е специално число:

- $16 / 2 = 8$ без остатък
- $16 / 4 = 4$ без остатък
- $16 / 1 = 16$ без остатък
- $16 / 8 = 2$ без остатък

Входни данни

Входът се чете от конзолата и се състои от **едно цяло число** в интервала [1...600000].

Изходни данни

Да се отпечатат на конзолата **всички специални числа**, разделени с **интервал**.

Примерен вход и изход

| Вход | Изход | Коментари |
|------|---|--|
| 3 | 1111 1113 1131 1133 1311 1313 1331 1333 3111 3113 3131 3133 3311 3313 3331 3333 | 3 / 1 = 3 без остатък 3 / 3 = 1 без остатък 3 / 3 = 1 без остатък 3 / 3 = 1 без остатък |

Насоки и подсказки

Решете задачата самостоятелно използвайки наученото от предишните две. Спомнете си разликата между операторите за **целочислено деление (/)** и **деление с остатък (%)** в C#.

Тестване в Judge системата

Тествайте решението си: <https://judge.softuni.bg/Contests/Practice/Index/515#3>.

Задача: цифри

Да се напише програма, която прочита от конзолата 1 цяло число в интервала [100...999], и след това го принтира определен брой пъти – модифицирайки го преди всяко принтиране по следния начин:

- Ако числото се дели на 5 без остатък, **извадете** от него **първата** му цифра.
- Ако числото се дели на 3 без остатък, **извадете** от него **втората** му цифра.
- Ако нито едно от горните условия не е вярно, **прибавете** към него **третата** му цифра.

Принтирайте на конзолата **N** **брой реда**, като всеки ред има **M** **на брой** числа, които са резултат от горните действия. Нека:

- **N** = **сбора** на **първата** и **втората** цифра на числото.
- **M** = **сбора** на **първата** и **третата** цифра на числото.

Входни данни

Входът се чете от **конзолата** и е **цяло** число в интервала [100...999].

Изходни данни

На конзолата да се отпечатат **всички цели числа**, които са резултат от дадените по-горе изчисления в съответния брой редове и колони, както в примерите.

Примерен вход и изход

| Вход | Изход | Коментари |
|------|--|---|
| 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 реда по 9 числа на всеки. Входното число 376 → нито на 5, нито на 3 → $376 + 6 = 382 \rightarrow$ нито на 5, нито на 3 → $382 + 6 = 388 + 6 = 394 + 6 = 400 \rightarrow$ деление на 5 → $400 - 3 = 397$ |

| Вход | Изход | Коментари |
|------|--|---|
| 132 | 129 126 123 120 119 121 123 120 119 121 123 120 | $(1 + 3) = 4$ и $(1 + 2) = 3 \rightarrow$ 4 реда по 3 числа на всеки ред Входното число 132 $132 \rightarrow$ деление на 3 → $132 - 3 = 129 \rightarrow$ деление на 3 → $129 - 3 = 126 \rightarrow$ деление на 3 → $126 - 3 = 123 \rightarrow$ деление на 3 → $123 - 3 = 120 \rightarrow$ деление на 5 → $120 - 1 = 119 \dots 121 \rightarrow$ нито на 5, нито на 3 → $121 + 2 = 123$ |

Насоки и подсказки

Решете задачата **самостоятелно**, използвайки наученото от предишните. Не забравяйте, че ще е нужно да дефинирате **отделна** променлива за всяка цифра на входното число.

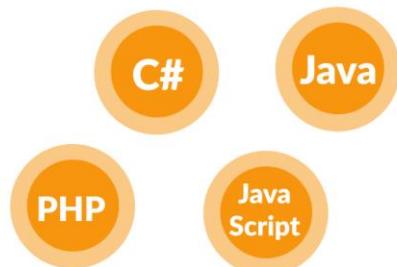
Тестване в Judge системата

Тествайте решението си: <https://judge.softuni.bg/Contests/Practice/Index/515#4>.

Качествено образование,
професия и работа за

Софтуерни инженери

- ✓ Безплатен старт за начинаещи - **присъствено и онлайн**
- ✓ Избор измежду **най-търсените професии** в софтуерната индустрия
- ✓ Съдействие за **кариерен старт**
- ✓ Отлични преподаватели, ментори и студентска общност



Софтуни предоставя съвременно иновативно образование за програмиране, ИТ и дигитални умения на хиляди млади хора. Програмата "**Софтуерен университет**" изгражда истински **професионалисти в света на програмирането**.

Включете се в цялостната програма по софтуерно инженерство на Софтуни, за да усвоите **най-търсените съвременни софтуерни технологии** чрез модерна учебна методика и с много практически задачи и проекти. **Учебният план** е разработен съвместно с ИТ фирмите и изгражда най-търсените от тях умения.

Софтуни работи пряко с компаниите от софтуерната индустрия, съдейтайки на своите студенти за реализацията им като успешни софтуерни инженери.

ПЪТЯТ НА СТУДЕНТА В СОФТУНИ



Кандидатствай

softuni.bg/apply

Глава 8.1. Подготовка за практически изпит – част I

В настоящата глава ще разгледаме няколко **задачи** с ниво на **трудност**, каквото може да очаквате от **задачите** на практическия **изпит** по “Основи на програмирането”. Ще преговорим и **упражнения** всички знания, които сте придобили от настоящата книга и през курса "Programing Basics".

Видео

Гледайте видео-урок по тази глава: <https://youtube.com/watch?v=Cw-75W5Az4I>.

Практически изпит по “Основи на програмирането”

Курсът "Programing Basics" приключва с **практически изпит**. Включени са **6** задачи, като ще имате **4 часа**, за да ги решите. **Всяка** от задачите на изпита ще **засяга** една от изучаваните **теми** по време на курса. Темите на задачите са както следва:

- Задача с прости сметки (без проверки).
- Задача с единична проверка.
- Задача с по-сложни проверки.
- Задача с единичен цикъл.
- Задача с вложени цикли (чертане на фигурука на конзолата).
- Задача с вложени цикли и по-сложна логика.

Система за онлайн оценяване (Judge)

Всички изпити и домашни се **тестват** автоматизирано през онлайн Judge система: <https://judge.softuni.bg>. За **всяка** от задачите има **открити** (нулеви) тестове, които ще ви помогнат да разберете какво се очаква от задачата и да поправите грешките си, както и **състезателни** тестове, които са **скрити** и проверяват дали задачата ви работи правилно. В Judge системата **се влиза** с вашия softuni.bg акаунт.

Как работи тестването в Judge системата? **Качвате** сорс кода и от менюто под него избирате да се компилира като **C#** програма. Програмата се **тества** с поредица тестове, като за всеки **успешен** тест получавате **точки**.

Задачи с прости пресмятания

Първата задача на практическия изпит по “Основи на програмирането” обхваща прости пресмятания без проверки и цикли. Ето няколко примера:

Задача: лице на триъгълник в равнината

Триъгълник в равнината е зададен чрез координатите на трите си върха. Първо е зададен **върхът** (x_1, y_1). След това са зададени останалите два върха: (x_2, y_2) и (x_3, y_3), които лежат на обща хоризонтална права (т.е. имат еднакви Y координати). Напишете програма, която пресмята **лицето** на триъгълника по координатите на трите му върха.

Вход

От конзолата се четат **6 цели** числа (по едно на ред): $x_1, y_1, x_2, y_2, x_3, y_3$.

- Всички входни числа са в диапазона [-1000...1000].
- Гарантирано е, че $y_2 = y_3$.

Изход

Да се отпечата на конзолата **лицето** на триъгълника.

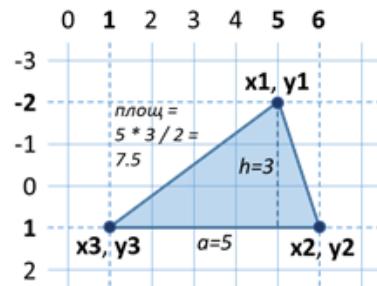
Примерен вход и изход

| Вход | Изход | Чертеж | Обяснения |
|-----------------------------|-------|--------|--|
| 5 -2 6 1 1 1 | 7.5 | | Страната на триъгълника: $a = 6 - 1 = 5$ Височината на триъгълника: $h = 1 - (-2) = 3$ Лицето на триъгълника: $S = a * h / 2 = 5 * 3 / 2 = 7.5$ |

| Вход | Изход | Чертеж | Обяснения |
|-------------------------------|-------|--------|---|
| 4 1 -1 -3 3 -3 | 8 | | Страната на триъгълника: $a = 3 - (-1) = 4$ Височината на триъгълника: $h = 1 - (-3) = 4$ Лицето на триъгълника: $S = a * h / 2 = 4 * 4 / 2 = 8$ |

Насоки и подсказки

Изключително важно при подобен тип задачи, при които се подават някакви координати, е да обърнем внимание на **реда**, в който се подават, както и правилно



да осмислим кои от координатите ще използваме и по какъв начин. В случая, на входа се подават $x_1, y_1, x_2, y_2, x_3, y_3$ в този си ред. Ако не спазваме тази последователност, решението става грешно. Първо пишем кода, който чете подадените данни:

```
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());
```

Трябва да пресметнем **страницата** и **височината** на триъгълника. От картинките, както и от условието $y_2 = y_3$ забелязваме, че едната **страница** винаги е успоредна на хоризонталната ос. Това означава, че нейната **дължина** е равна на дълчината на отсечката между нейните координати **x_2 и x_3** , която е равна на разликата между по-голямата и по-малката координата. Аналогично можем да изчислим и **височината**. Тя винаги ще е равна на разликата между **y_1 и y_2** (или y_3 , тъй като са равни). Тъй като не знаем дали винаги x_2 ще е по-голям от x_3 , или y_1 ще е под или над страницата на триъгълника, ще използваме **абсолютните стойности** на разликата, за да получаваме винаги положителни числа, понеже една отсечка не може да има отрицателна дължина.

```
int a = Math.Abs(x2 - x3);
int h = Math.Abs(y2 - y1);
```

По познатата ни от училище формула за намиране на **лице на триъгълник** ще пресметнем лицето. Важно нещо, което трябва да съобразим, е въпреки че на входа получаваме само цели числа, **лицето** не винаги ще е цяло число. Затова за лицето използваме променлива от тип **double**. Налага се да конвертираме и дясната страница на уравнението, понеже ако подадем цели числа като параметри на уравнението, резултатът ни също ще е цяло число.

```
double s = (double)a * h / 2;
```

Единственото, което остава, е да отпечатаме лицето на конзолата.

```
Console.WriteLine(s);
```

Тестване в Judge системата

Тествайте решението си: <https://judge.softuni.bg/Contests/Practice/Index/516#0>.

Задача: пренасяне на тухли

Строителни работници трябва да пренесат общо **x тухли**. **Работниците** са **w** на брой и работят едновременно. Те превозват тухлите в колички, всяка с **вместимост m** тухли. Напишете програма, която прочита числата **x , w и m** и пресмята колко най-малко курса трябва да направят работниците, за да превозят тухлите.

Вход

От конзолата се четат **3 цели числа** (по едно на ред):

- **Броят тухли x** се чете от първия ред.
- **Броят работници w** се чете от втория ред.
- **Вместимостта на количката m** се чете от третия ред.

Всички входни числа са цели и в диапазона [1...1000].

Изход

Да се отпечатат на конзолата **минималният брой курсове**, необходими за превозване на тухлите.

Примерен вход и изход

| Вход | Изход | Обяснения |
|----------------|-------|--|
| 120 2 30 | 2 | Имаме 2 работника, всеки вози по 30 тухли на курс. Общо работниците возят по 60 тухли на курс. За да превозят 120 тухли, са необходими точно 2 курса. |

| Вход | Изход | Обяснения |
|----------------|-------|---|
| 355 3 10 | 12 | Имаме 3 работника, всеки вози по 10 тухли на курс. Общо работниците возят по 30 тухли на курс. За да превозят 355 тухли, са необходими точно 12 курса: 11 пълни курса превозват 330 тухли и последният 12-ти курс пренася последните 25 тухли. |

| Вход | Изход | Обяснения |
|---------------|-------|--|
| 5 12 30 | 1 | Имаме 5 работника, всеки вози по 30 тухли на курс. Общо работниците возят по 150 тухли на курс. За да превозят 5 тухли, е достатъчен само 1 курс (макар и непълен, само с 5 тухли). |

Насоки и подсказки

Входът е стандартен, като единствено трябва да внимаваме за последователността, в която прочитаме данните.

```
int x = int.Parse(Console.ReadLine());
int w = int.Parse(Console.ReadLine());
int m = int.Parse(Console.ReadLine());
```

Пресмятаме колко **тухли** носят работниците на един курс.

```
int bricksInOneCourse = w * m;
```

Като разделим общия брой на тухлите, пренесени за **1 курс**, ще получим броя **курсове**, необходими за пренасянето им. Трябва да съобразим, че при деление на цели числа се пренебрегва остатъка и се закръгли винаги надолу.

За да избегнем това ще конвертираме дясната страна на уравнението към **double** и ще използваме функцията **Math.Ceiling(...)**, за да закръглим получения резултат винаги нагоре. Когато тухлите могат да се пренесат с **точен брой курсове**, делението ще връща точно число и няма да има нищо за закръгляне. Съответно, когато не е така, резултатът от делението ще е **броя на точните курсове**, но с десетична част. Десетичната част ще се закръгли нагоре и така ще се получи нужният **1 курс** за оставащите тухли.

```
double totalCourses = Math.Ceiling((double)x / bricksInOneCourse);
```

Накрая принтираме резултата на конзолата.

```
Console.WriteLine(totalCourses);
```

Тестване в Judge системата

Тествайте решението си: <https://judge.softuni.bg/Contests/Practice/Index/516#1>.

Задачи с единична проверка

Втората задача на практическия изпит по “Основи на програмирането” обхваща условна конструкция и прости премятания. Ето няколко примера:

Задача: точка върху отсечка

Върху хоризонтална прива е разположена **хоризонтална отсечка**, зададена с **x** координатите на двата си края: **first** и **second**. Точка е разположена **върху** същата хоризонтална прива и е зададена с **x** координатата си. Напишете програма, която проверява дали точката е **вътре или вън** от отсечката и изчислява **разстоянието до по-близкия край** на отсечката.

Вход

От конзолата се четат **3 цели числа** (по едно на ред):

- На първия ред стои числото **first** – **единния край на отсечката**.
- На втория ред стои числото **second** – **другия край на отсечката**.
- На третия ред стои числото **point** – **местоположението на точката**.

Всички входни числа са цели и в диапазона [-1000...1000].

Изход

Резултатът да се отпечата на конзолата:

- На първия ред да се отпечата "in" или "out" – дали точката е върху отсечката или извън нея.
- На втория ред да се отпечата разстоянието от точката до най-близкия край на отсечката.

Примерен вход и изход

| Вход | Изход | Визуализация |
|--------------|---------|---|
| 10 5 7 | in 2 |  |

| Вход | Изход | Визуализация |
|--------------|----------|---|
| 8 10 5 | out 3 |  |

| Вход | Изход | Визуализация |
|--------------|----------|---|
| 1 -2 3 | out 2 |  |

Насоки и подсказки

Четем входа от конзолата:

```
int first = int.Parse(Console.ReadLine());
int second = int.Parse(Console.ReadLine());
int point = int.Parse(Console.ReadLine());
```

Тъй като не знаем коя **точка** е от ляво и коя е от дясно, ще си направим две променливи, които да ни пазят това. Тъй като **левата точка** е винаги тази с по-малката **x координата**, ще ползваме **Math.Min(...)**, за да я намерим. Съответно, **дясната** е винаги тази с по-голяма **x координата** и ползваме **Math.Max(...)**. Ще намерим и разстоянието от **точката x** до **двете точки**. Понеже не знаем положението им една спрямо друга, ще използваме **Math.Abs(...)**, за да получим положителен резултат.

```
int left = Math.Min(first, second);
int right = Math.Max(first, second);

int distanceLeft = Math.Abs(left - point);
int distanceRight = Math.Abs(right - point);
```

По-малкото от двете разстояния ще намерим ползвайки **Math.Min(...)**.

```
int minDistance = Math.Min(distanceLeft, distanceRight);
```

Остава да намерим дали точката е на линията или извън нея.

- Точката ще се намира **на линията** винаги, когато тя **съвпада** с някоя от другите две точки или **x координатата ѝ** се намира между тях.
- В противен случай, точката се намира **извън линията**.

След проверката печатаме едното от двете съобщения, спрямо това коя проверка е удовлетворена.

```
if (point >= left && point <= right)
{
    Console.WriteLine("in");
}
else
{
    Console.WriteLine("out");
}
```

Накрая принтираме **разстоянието**, намерено преди това.

```
Console.WriteLine(minDistance);
```

Тестване в Judge системата

Тествайте решението си: <https://judge.softuni.bg/Contests/Practice/Index/516#2>.

Задача: точка във фигура

Да се напише програма, която проверява дали дадена точка (с координати **x** и **y**) е **вътре** или **извън** фигурата, която е дадена на картинката вдясно.

Примерен вход и изход

| Вход | Изход |
|---------|-------|
| 8 -5 | in |

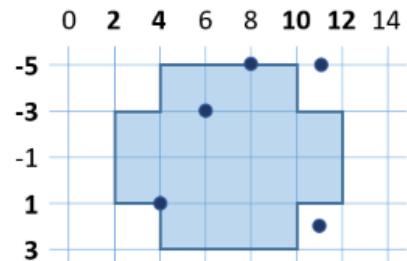
| Вход | Изход |
|---------|-------|
| 6 -3 | in |

| Вход | Изход |
|----------|-------|
| 11 -5 | out |

| Вход | Изход |
|---------|-------|
| 11 2 | out |

Вход

От конзолата се четат **две цели числа** (по едно на ред): **x** и **y**. Всички входни числа са цели и в диапазона [-1000...1000].

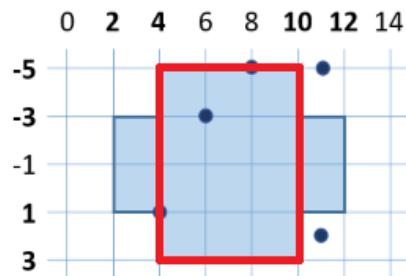
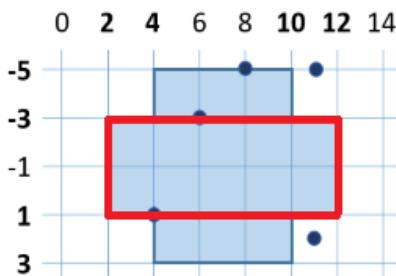


Изход

Да се отпечата на конзолата "in" или "out" – дали точката е **вътре** или **извън** фигурана (на контура е вътре).

Насоки и подсказки

За да разберем дали **точката** е във фигурана, ще разделим **фигурата** на два четириъгълника:



Достатъчно условие е **точката** да се намира в един от тях, за да се намира във **фигурата**. Четем от конзолата входните данни:

```
int x = int.Parse(Console.ReadLine());
int y = int.Parse(Console.ReadLine());
```

Ще създадем две променливи, които ще отбелязват дали **точката** се намира в някой от правоъгълниците.

```
bool pointInRect1 = x >= 2 && x <= 12 && y >= -3 && y <= 1;
bool pointInRect2 = x >= 4 && x <= 10 && y >= -5 && y <= 3;
```

При отпечатването на съобщението ще проверим дали някоя от променливите е приела стойност **true**. Достатъчно е **само една** от тях да е **true**, за да се намира точката във фигурана.

```
if (pointInRect1 || pointInRect2)
{
    Console.WriteLine("in");
}
else
{
    Console.WriteLine("out");
}
```

Тестване в Judge системата

Тествайте решението си: <https://judge.softuni.bg/Contests/Practice/Index/516#3>.

Задачи с по-сложни проверки

Третата задача на практическия изпит по “Основи на програмирането” включва няколко вложени проверки съчетани с прости пресмятания. Ето няколко примера:

Задача: дата след 5 дни

Дадени са две числа **d** (ден) и **m** (месец), които формират **дата**. Да се напише програма, която отпечатва датата, която ще бъде **след 5 дни**. Например 5 дни след 28.03 е датата 2.04. Приемаме, че месеците: април, юни, септември и ноември имат по 30 дни, февруари има 28 дни, а останалите имат по 31 дни. Месеците да се отпечатват с **водеща нула**, когато са едноцифreni (например 01, 08).

Вход

Входът се чете от конзолата и се състои от два реда:

- На първия ред стои едно цяло число **d** в интервала [1...31] – ден. Номерът на деня не надвишава броя дни в съответния месец (напр. 28 за февруари).
- На втория ред стои едно цяло число **m** в интервала [1...12] – месец. Месец 1 е януари, месец 2 е февруари, ..., месец 12 е декември. Месецът може да съдържа водеща нула (напр. април може да бъде изписан като 4 или 04).

Изход

Отпечатайте на конзолата един единствен ред, съдържащ дата след 5 дни във формат **ден.месец**. Месецът трябва да бъде двуцифreno число с водеща нула, ако е необходимо. Денят трябва да е без водеща нула.

Примерен вход и изход

| Вход | Изход | Вход | Изход | Вход | Изход | Вход | Изход |
|----------|-------|----------|-------|---------|-------|----------|-------|
| 28 03 | 2.04 | 27 12 | 1.01 | 25 1 | 30.01 | 26 02 | 3.03 |

Насоки и подсказки

Четем си входа от конзолата.

```
int d = int.Parse(Console.ReadLine());
int m = int.Parse(Console.ReadLine());
```

За да си направим по-лесно проверките, ще си създадем една променлива, която ще съдържа **броя дни**, които има в месеца, който сме задали.

```
int daysInMonth = 31;
if (m == 2)
{
    daysInMonth = 28;
```

```

}
if (m == 4 || m == 6 || m == 9 || m == 11)
{
    daysInMonth = 30;
}

```

Добавяме към **дения** 5 дни: `d += 5;`

Проверяваме дали **деният** не е станал по-голям от броя дни, които има в съответния **месец**. Ако това е така, трябва да извадим дните от месеца от получния ден, за да получим нашият ден на кой ден от следващия месец съответства.

```

if (d > daysInMonth)
{
    d -= daysInMonth;
}

```

След като сме минали в **следващия месец**, това трябва да се отбележи, като увеличим първоначално зададения с 1. Трябва да проверим, дали той не е станал по-голям от 12 и ако е така, да коригираме. Тъй като няма как да прескочим повече от **един месец**, когато увеличаваме с 5 дни, долната проверка е достатъчна.

```

if (d > daysInMonth)
{
    d -= daysInMonth;
    m++;
    if (m > 12)
    {
        m = 1;
    }
}

```

Остава само да принтираме резултата на конзолата. Важно е да **форматираме изхода** правилно, за да се появява водещата нула в първите 9 месеца. Това става, като добавим **форматиращ стринг :D2** след втория елемент.

```
Console.WriteLine("{0}.{1:D2}", d, m);
```

Тестване в Judge системата

Тествайте решението си: <https://judge.softuni.bg/Contests/Practice/Index/516#4>.

Задача: суми от 3 числа

Дадени са 3 цели числа. Да се напише програма, която проверява дали **сумата на две от числата е равна на третото**. Например, ако числата са 3, 5 и 2, сумата на две от числата е равна на третото: $2 + 3 = 5$.

Вход

От конзолата се четат **три цели числа**, по едно на ред, в диапазона [1...1000].

Изход

- Да се отпечата на конзолата един ред, съдържащ решението на задачата във формат " $a + b = c$ ", където a , b и c са измежду входните три числа и $a \leq b$.
- Ако задачата няма решение, да се отпечата "No".

Примерен вход и изход

| Вход | Изход | Вход | Изход | Вход | Изход | Вход | Изход |
|------|-------------|------|-------------|------|-------|------|-------|
| 3 | | 2 | | 1 | | 2 | |
| 5 | $2 + 3 = 5$ | 2 | $2 + 2 = 4$ | 1 | No | 6 | |
| 2 | | 4 | | 5 | | 3 | No |

Насоки и подсказки

Приемаме си входа от конзолата.

```
int a = int.Parse(Console.ReadLine());
int b = int.Parse(Console.ReadLine());
int c = int.Parse(Console.ReadLine());
```

Трябва да проверим дали **сумата** на някоя двойка числа е равна на третото. Имаме три възможни случая:

- $a + b = c$
- $a + c = b$
- $b + c = a$

Ще си напишем **рамка**, която после ще допълним с нужния код. Ако никое от горните три условия не е изпълнено, ще зададем на програмата да принтира "No".

```
if (a + b == c)
{
    // TODO
}
else if (a + c == b)
{
    // TODO
}
else if (b + c == a)
{
    // TODO
}
```

```
else
{
    Console.WriteLine("No");
}
```

Сега остава да разберем реда, в който ще се изписват **двете събираеми** на изхода на програмата. За целта ще направим **вложено условие**, което проверява кое от двете числа е по-голямото. При първия случай, ще стане по този начин:

```
if (a + b == c)
{
    if (a > b)
    {
        Console.WriteLine("{0} + {1} = {2}", b, a, c);
    }
    else
    {
        Console.WriteLine("{0} + {1} = {2}", a, b, c);
    }
}
```

Аналогично, ще допълним и другите два случая. Пълният код на проверките и изходът на програмата ще изглежда така:

```
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");
}

```

Тестване в Judge системата

Тествайте решението си: <https://judge.softuni.bg/Contests/Practice/Index/516#5>.

Задачи с единичен цикъл

Четвъртата задача на практическия изпит по “Основи на програмирането” включва единичен цикъл с прости логики в него. Ето няколко примера:

Задача: суми през 3

Дадени са n цели числа a_1, a_2, \dots, a_n . Да се пресметнат сумите:

- $\text{sum1} = a_1 + a_4 + a_7 + \dots$ (сумират се числата, започвайки от първото със стъпка 3).
- $\text{sum2} = a_2 + a_5 + a_8 + \dots$ (сумират се числата, започвайки от второто със стъпка 3).
- $\text{sum3} = a_3 + a_6 + a_9 + \dots$ (сумират се числата, започвайки от третото със стъпка 3).

Вход

Входните данни се четат от конзолата.

- На първия ред стои цяло числото n ($0 \leq n \leq 1000$).

- На следващите n реда стоят n цели числа в интервала [-1000...1000]: a_1, a_2, \dots, a_n .

Изход

На конзолата трябва да се отпечатат 3 реда, съдържащи търсените 3 суми, във формат като в примерите.

Примерен вход и изход

| Вход | Изход | Вход | Изход | Вход | Изход |
|-------------|----------------------------------|-------------------------|------------------------------------|----------------------------|------------------------------------|
| 2 3 5 | sum1 = 3 sum2 = 5 sum3 = 0 | 4 7 -2 6 12 | sum1 = 19 sum2 = -2 sum3 = 6 | 5 3 5 2 7 8 | sum1 = 10 sum2 = 13 sum3 = 2 |
| | | | | | |

Насоки и подсказки

Ще вземем **броя на числата** от конзолата и ще декларираме **начални стойности** на трите суми.

```
int n = int.Parse(Console.ReadLine());
```

```
int sum1 = 0;
int sum2 = 0;
int sum3 = 0;
```

Тъй като не знаем предварително колко числа ще обработваме, ще си ги взимаме едно по едно в **цикъл**, който ще се повтори **n на брой пъти** и ще ги обработваме в тялото на цикъла.

```
for (int i = 0; i < n; i++)
{
    int a = int.Parse(Console.ReadLine());
    //TODO
}
```

За да разберем в коя от **трите суми** трябва да добавим числото, ще разделим **поредния му номер на три** и ще използваме **остатъка**. Ще използваме променливата **i**, която следи **броя завъртания** на цикъла, за да разберем на кое поред число сме. Когато остатъкът от **i/3** е **нула**, това означава, че ще добавяме това число към **първата** сума, когато е **1** към **втората** и когато е **2** към **третата**.

```

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;
    }
}

```

Накрая, ще отпечатаме резултата на конзолата в изисквания **формат**.

```

Console.WriteLine("sum1 = {0}", sum1);
Console.WriteLine("sum1 = {0}", sum2);
Console.WriteLine("sum1 = {0}", sum3);

```

Тестване в Judge системата

Тествайте решението си: <https://judge.softuni.bg/Contests/Practice/Index/516#6>.

Задача: поредица от нарастващи елементи

Дадена е редица от n числа: a_1, a_2, \dots, a_n . Да се пресметне **дължината на най-дългата нарастваща поредица** от последователни елементи в редицата от числа.

Вход

Входните данни се четат от конзолата.

- На първия ред стои цяло число n ($0 \leq n \leq 1000$).
- На следващите n реда стоят n цели числа в интервала $[-1000...1000]$: a_1, a_2, \dots, a_n .

Изход

На конзолата трябва да се отпечата едно число – **дължината на най-дългата нарастваща редица**.

Примерен вход и изход

| Вход | Изход | Вход | Изход | Вход | Изход | Вход | Изход |
|------|-------|------|-------|------|-------|------|-------|
| 3 | | 4 | | 4 | | 4 | |
| 5 | 2 | 2 | | 1 | | 5 | |
| 2 | | 8 | 2 | 2 | 3 | 6 | 4 |
| 4 | | 7 | | 4 | | 7 | |
| | | 6 | | 4 | | 8 | |

Насоки и подсказки

За решението на тази задача трябва да помислим малко **по-алгоритично**. Дадена ни е **редица от числа** и трябва да проверяваме дали всяко **следващо**, ще бъде **по-голямо от предното** и ако е така да броим колко дълга е редицата, в която това условие е изпълнено. След това трябва да намерим **коя редица** от всички такива е **най-дълга**. За целта, нека да си направим няколко променливи, които ще ползваме през хода на задачата.

```
int n = int.Parse(Console.ReadLine());
int countCurrentLongest = 0;
int countLongest = 0;
int aPrev = 0;
int a = 0;
```

Променливата **n** е **броя числа**, които ще получим от конзолата. В **countCurrentLongest** ще запазваме **броя на елементите** в нарастващата редица, която **броим в момента**. Например при редицата (5, 6, 1, 2, 3) **countCurrentLongest** ще бъде 2, когато сме стигнали **втория елемент** от броенето (5, 6, 1, 2, 3) и ще стане 3, когато стигнем **последния елемент** (5, 6, 1, 2, 3), понеже нарастващата редица 1, 2, 3 има 3 елемента. Ще използваме **countLongest**, за да запазим търсената в задачата **най-дълга** нарастваща редица. Останалите променливи са **a** – **числото, на което се намираме в момента**, и **aPrev** – **предишното число**, което ще сравним с **a**, за да разберем дали редицата **расте**.

Започваме да въртим числата и проверяваме дали настоящото число **a** е по-голямо от предходното **aPrev**. Ако това е изпълнено, значи редицата **е нарастваща** и трябва да увеличим броя ѝ с **1**. Това запазваме в променливата, която следи дължината на редицата, в която се намираме в момента, а именно – **countCurrentLongest**. Ако числото **a не е по-голямо** от предходното, това означава, че започва нова редица и трябва да стартираме броенето от **1**. Накрая, след всички проверки, **aPrev** става **числото**, което използваме **в момента**, и започваме цикъла от начало със **следващото** въведено **a**.

Ето и примерна реализация на описания алгоритъм:

```

for (int i = 0; i < n; i++)
{
    a = int.Parse(Console.ReadLine());

    if (a > aPrev)
    {
        countCurrentLongest++;
    }
    else
    {
        countCurrentLongest = 1;
    }

    aPrev = a;
}

```

Остава да разберем коя от всички редици е **най-дълга**. Това ще направим с проверка в цикъла дали **редицата**, в която се намираме **в момента**, е станала по-дълга от дълчината на **най-дългата намерена до сега**. Целият цикъл ще изглежда така:

```

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;
}

```

Накрая принтираме дълчината на **най-дългата** намерена редица.

```
Console.WriteLine(countLongest);
```

Тестване в Judge системата

Тествайте решението си: <https://judge.softuni.bg/Contests/Practice/Index/516#7>.

Задачи за чертане на фигурки на конзолата

Петата задача на практическия изпит по “Основи на програмирането” изисква използване на един или няколко вложени цикъла за рисуване на някаква фигурка на конзолата. Може да се изискват логически размишления, извършване на прости пресмятания и проверки. Задачата проверява способността на студентите да мислят логически и да измислят прости алгоритми за решаване на задачи, т.е. да мислят алгоритмично. Ето няколко примера за изпитни задачи:

Задача: перфектен диамант

Да се напише програма, която прочита от конзолата цяло число n и чертае **перфектен диамант** с размер n като в примерите по-долу.

Примерен вход и изход

| Вход | Изход | Вход | Изход | Вход | Изход | Вход | Изход |
|------|------------------------|------|--|------|--|------|--|
| 2 | <pre> * * - * * </pre> | 3 | <pre> * * - * * - * - * * - * * </pre> | 4 | <pre> * * - * * - * - * * - * - * - * * - * - * * - * * </pre> | 5 | <pre> * * - * * - * - * * - * - * - * * - * - * - * * - * - * * - * * </pre> |

Вход

Входът е цяло число n в интервала [1...1000].

Изход

На конзолата трябва да се отпечата диамантът като в примерите.

Насоки и подсказки

В задачите с чертане на фигурки най-важното, което трябва да преценим е **последователността**, в която ще рисуваме. Кои елементи се **повтарят** и с какви **стъпки**. Ясно може да забележим, че **горната и долната** част на диаманта са **еднакви**. Най-лесно ще решим задачата, като направим **един цикъл**, който чертае **горната** част, и след това още **един**, който чертае **долната** (обратно на горната).

Ще си прочетем числото **n** от конзолата.

```
int n = int.Parse(Console.ReadLine());
```

Започваме да рисуваме горната половина на диаманта. Ясно виждаме, че **всеки ред** започва с няколко **празни места и ***. Ако се вгледаме по- внимателно, ще забележим, че **празните места** са винаги равни на **n - броя на реда** (на първия ред са $n-1$, на втория – $n-2$ и т.н.) Ще започнем с това да нарисуваме броя **празни места**, както и **първата звездинка**. Нека не забравяме да използваме **Console.WriteLine(...)** вместо **Console.WriteLine(...)**, за да оставаме на **същия** ред. На края на реда пишем **Console.WriteLine(...)**, за да преминем на **нов** ред. Забележете, че започваме да броим от **1**, а не от **0**. След това ще остане само да добавим няколко пъти **-***, за да довършим реда.

Ето фрагмент от кода за начертаване на горната част на диаманта:

```
for (int i = 1; i <= n; i++)
{
    Console.Write(new string(' ', n - i));
    Console.WriteLine("*");
    // TODO: Draw the rest of the line
    Console.WriteLine();
}
```

Остава да довършим **всеки** ред с нужния брой **-*** елементи. На всеки ред трябва да добавим **i - 1** такива **елемента** (на първия $1-1 \rightarrow 0$, на втория $\rightarrow 1$ и т.н.).

Ето и пълният код за начертаване на горната част на диаманта:

```
for (int i = 1; i <= n; i++)
{
    Console.Write(new string(' ', n - i));
    Console.WriteLine("*");
    for (int j = 0; j < i - 1; j++)
    {
        Console.WriteLine("-*");
    }
    Console.WriteLine();
}
```

За да изрисуваме **долната част** на диаманта, трябва да обърнем **горната** на обратно. Ще броим от **n - 1**, тъй като ако започнем от **n**, ще изрисуваме средния ред два пъти. Не забравяйте да смените **стъпката** от **++** на **--**.

Ето го и кода за начертаване на **долната част на диаманта**:

```

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();
}

```

Остава да си сглобим цялата програма като първо четем входа, печатаме горната част на диаманта и след него и долната част на диаманта.

Тестване в Judge системата

Тествайте решението си: <https://judge.softuni.bg/Contests/Practice/Index/516#8>.

Задача: правоъгълник със звездички в центъра

Да се напише програма, която прочита от конзолата цяло число **n** и чертае правоъгълник с размер **n** с две звездички в центъра, като в примерите по-долу.

Примерен вход и изход

| Вход | Изход | Вход | Изход | Вход | Изход | Вход | Изход |
|------|--------------------|------|--|------|--|------|--|
| 2 | %%% %**% %%% | 3 | %%%%%% % % % ** % % % %%%%%% | 4 | %%%%%% % % % ** % % % %%%%%% | 5 | %%%%%% % % % ** % % % %%%%%% |

Вход

Входът е цяло число **n** в интервала [2...1000].

Изход

На конзолата трябва да се отпечата правоъгълникът като в примерите.

Насоки и подсказки

Прочитаме входните данни от задачата.

```
int n = int.Parse(Console.ReadLine());
```

Първото нещо, което лесно забелязваме, е че **първият и последният ред** съдържат **$2 * n$** символа **%**. Ще започнем с това и после ще нарисуваме средата на четириъгълника.

```
Console.WriteLine(new string('%', n * 2));
// TODO: Draw middle of the rectangle
Console.WriteLine(new string('%', n * 2));
```

От дадените примери виждаме, че **средата** на фигурата винаги има **нечетен брой** редове. Забелязваме, че когато е зададено **четно число**, броят на редовете е равен на **предишното нечетно** ($2 \rightarrow 1$, $4 \rightarrow 3$ и т.н.).

Следователно, можем да си създадем променлива, която държи броя редове, които ще има нашият правоъгълник, и да я коригираме, ако числото **n** е **четно**.

След това ще нарисуваме **правоъгълника без звездичките**. Всеки ред има за **начало и край** символа **%** и между тях **$2 * n - 2$** празни места (ширината е **$2 * n$** и вадим 2 за двета процента в края).

Не забравяйте да преместите кода за **последния ред** след цикъла.

Ето примерна реализация:

```
int numRows = n;
if (n % 2 == 0)
{
    numRows--;
}
for (int i = 0; i < numRows; i++)
{
    Console.Write("%");
    Console.Write(new string(' ', n * 2 - 2));
    // TODO: Place the stars
    Console.Write("%");
    Console.WriteLine();
}
```

Можем да стапираме и тестваме кода до тук. Всичко без двете звездички в средата трябва да работи коректно.

Сега остава **в тялото** на цикъла да добавим и **звездичките**. Ще направим проверка дали сме на **средния ред**. Ако сме на средния, ще рисуваме **реда** заедно **със звездичките**, ако не – ще рисуваме **нормален ред**. Редът със звездичките се състои от **$n - 2$** **празни места** (**n** е половината дължина и махаме звездичката и процента), **две звезди** и **ново $n - 2$ празни места**. Двета процента **%** в началото и в края на реда си ги оставяме извън проверката.

Ето и **примерен код**, който имплементира описаната идея:

```
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.Write("%");
    Console.WriteLine();
}
```

Тестване в Judge системата

Тествайте решението си: <https://judge.softuni.bg/Contests/Practice/Index/516#9>.

Задачи с вложени цикли с по-сложна логика

Последната (шеста) задача от практическия изпит по “Основи на програмирането” изисква използване на **няколко вложени цикъла и по-сложна логика в тях**. Задачата проверява способността на студентите да мислят алгоритично и да решават нетривиални задачи, изискаващи съставянето на цикли. Следват няколко примера за изпитни задачи.

Задача: четворки нарастващи числа

По дадена двойка числа **a** и **b** да се генерират всички четворки **n1, n2, n3, n4**, за които $a \leq n1 < n2 < n3 < n4 \leq b$.

Вход

Входът съдържа две цели числа **a** и **b** в интервала [0...1000], по едно на ред.

Изход

Изходът съдържа търсените четворки числа, в нарастващ ред, по една на ред.

Примерен вход и изход

| Вход | Изход | Вход | Изход | Вход | Изход |
|--------|---|--------|-------|----------|-------------|
| 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 |

Насоки и подсказки

Ще прочетем входните данни от конзолата. Създаваме и допълнителната променлива **count**, която ще следи дали има съществуваща редица числа.

```
int a = int.Parse(Console.ReadLine());
int b = int.Parse(Console.ReadLine());
int count = 0;
```

Най-лесно ще решим задачата, ако логически я разделим на части. Ако се изиска да изведем всички редици от едно число между **a** и **b**, ще го направим с един цикъл, който изкарва всички числа от **a** до **b**. Нека помислим как ще стане това с редици от две числа. Отговорът е лесен – ще ползваме вложени цикли.

```
for (int i = a; i <= b; i++)
{
    for (int j = i + 1; j <= b; j++)
    {
        Console.WriteLine("{0} {1}", i, j);
    }
}
```

Можем да тестваме недописаната програма, за да проверим дали е вярна до този момент. Тя трябва да отпечата всички двойки числа **i, j**, за които **i ≤ j**.

Тъй като всяко **следващо число** от редицата трябва да е **по-голямо** от **предишното**, вторият цикъл ще се върти от **i + 1** (следващото по-голямо число). Съответно, ако **не съществува редица** от две нарастващи числа (**a** и **b** са равни), вторият цикъл няма да се изпълни и няма да се разпечата нищо на конзолата.

Аналогично, остава да реализираме по същия начин **вложени цикли** и за четири числа. Ще добавим и **брояча**, който инициализираме в началото, за да знаем дали **съществува такава редица**.

```
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);
            }
        }
    }
}
```

```

    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++;
        }
    }
}

```

Накрая ще проверим дали броячът е равен на 0 и съответно ще принтираме “No” на конзолата, ако е така.

```

if (count == 0)
{
    Console.WriteLine("No");
}

```

Тестване в Judge системата

Тествайте решението си: <https://judge.softuni.bg/Contests/Practice/Index/516#10>.

Задача: генериране на правоъгълници

По дадено число **n** и **минимална площ m** да се генерират всички правоъгълници с цели координати в интервала $[-n...n]$ с площ поне **m**. Генерираните правоъгълници да се отпечатат в следния формат:

$(\text{left}, \text{top}) (\text{bottom}, \text{right}) \rightarrow \text{area}$

Правоъгълниците се задават чрез горния си ляв и долния си десен ъгъл. В сила са следните неравенства:

- $-n \leq \text{left} < \text{right} \leq n$
- $-n \leq \text{top} < \text{bottom} \leq n$

Вход

От конзолата се въвеждат **две числа**, по едно на ред:

- Цяло число **n** в интервала $[1...100]$ – задава минималната и максималната координата на връх.
- Цяло число **m** в интервала $[0...50000]$ – задава минималната площ на генерираните правоъгълници.

Изход

- На конзолата трябва да се отпечатат описаните правоъгълници във формат като в примерите по-долу.
- Ако за числата **n** и **m** няма нито един правоъгълник, да се изведе “**No**”.
- Редът на извеждане на правоъгълниците е без значение.

Примерен вход и изход

| Вход | Изход | Вход | Изход | Вход | Изход |
|--------|--|---------|-------|---------|-----------------------|
| 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 |
| | | | | | |

Насоки и подсказки

Да прочетем входните данни от конзолата. Ще създадем един **брояч** и една **променлива**, която ще използваме за пресмятането на **площта на правоъгълника**.

```
int n = int.Parse(Console.ReadLine());
int m = int.Parse(Console.ReadLine());
int count = 0;
int area = 0;
```

Изключително важно е да успеем да си представим задачата, преди да започнем да я решаваме. В нашия случай се изиска да търсим правоъгълници с различни размери в координатна система. Нещото, което знаем е, че **левата точка** винаги ще има координата **x**, **по-малка от дясната**. Съответно **горната** винаги ще има **по-малка** координата **y** от **долната**. За да намерим всички правоъгълници, ще трябва да направим няколко вложени **цикъла**, подобно на тези от предходната задача, но този път **не всеки следващ цикъл** ще започва от **следващото число**, защото някои от координатите може да са **равни** (например **left** и **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
            }
        }
    }
}
```

```

    }
}
}
```

С променливите **left** и **right** ще следим координатите по **хоризонталата**, а с **top** и **bottom** – по **вертикалата**. Важното тук е да знаем кои координати кои са, за да можем да изчислим правилно страните на четириъгълника. Сега трябва да намерим **лицето на правоъгълника** и да направим проверка дали то е **по-голямо** или **равно** на **m**. Едната страна ще е **разликата между left и right**, а другата – **между top и bottom**. Тъй като координатите евентуално може да са разменени, ще ползваме **абсолютни стойности**. Отново добавяме и **брояча** в цикъла, като броим **само четириъгълниците**, които изписваме. Важно е да забележим, че поредността на изписване е **left, top, right, bottom**, тъй като така е зададено в условието.

Ето го и кода на основната част от решението на задачата:

```

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++;
                }
            }
        }
    }
}
```

Накрая принтираме “**No**”, ако не съществуват такива правоъгълници.

```

if (count == 0)
{
    Console.WriteLine("No");
}
```

Тестване в Judge системата

Тествайте решението си: <https://judge.softuni.bg/Contests/Practice/Index/516#11>.

Глава 8.2. Подготовка за практически изпит – част II

В настоящата глава ще разгледаме един **практически изпит по основи на програмирането**, проведен в СофтУни на 18 декември 2016 г. Задачите дават добра представа какво можем да очакваме на приемния изпит по програмиране в СофтУни. Изпитът покрива изучавания учебен материал от настоящата книга и от курса "Programming Basics" в СофтУни.

Видео

Гледайте видео-урок по тази глава: <https://youtube.com/watch?v=Z-Sq4--FoGI>.

Изпитни задачи

Традиционно приемният изпит в СофтУни се състои от **6 практически задачи по програмиране**:

- Задача с прости сметки (без проверки).
- Задача с единична проверка.
- Задача с по-сложни проверки.
- Задача с единичен цикъл.
- Задача с вложени цикли (чертане на фигурука на конзолата).
- Задача с вложени цикли и по-сложна логика.

Да разгледаме една **реална изпитна тема**, задачите в нея и решенията им.

Задача: разстояние

Напишете програма, която да пресмята колко километра изминава кола, за която знаем първоначалната скорост (км/ч), времето в минути, след което **увеличава** скоростта с 10%, **второ време**, след което **намалява** скоростта с 5%, и времето до **края** на пътуването. За да намерите разстоянието трябва да **превърнете минутите в часове** (например 70 минути = 1.1666 часа).

Входни данни

От конзолата се четат **4 реда**:

- Първоначалната скорост в км/ч – цяло число в интервала [1...300].
- Първото време в минути – цяло число в интервала [1...1000].
- Второто време в минути – цяло число в интервала [1...1000].
- Третото време в минути – цяло число в интервала [1...1000].

Изходни данни

Да се отпечата на конзолата **едно число: изминатите километри**, форматирани до втория символ след десетичния знак.

Примерен вход и изход

| Вход | Изход | Обяснения |
|-------------------------|--------|---|
| 90 60 70 80 | 330.90 | Разстояние с първоначална скорост: $90 \text{ км/ч} * 1 \text{ час} (60 \text{ мин}) = 90 \text{ км}$ След увеличението: $90 + 10\% = 99.00 \text{ км/ч} * 1.166 \text{ часа} (70 \text{ мин}) = 115.50 \text{ км}$ След намаляването: $99 - 5\% = 94.05 \text{ км/ч} * 1.33 \text{ часа} (80 \text{ мин}) = 125.40 \text{ км}$ Общо изминати: 330.9 км |
| 140 112 75 190 | 917.12 | Разстояние с първоначална скорост: $140 \text{ км/ч} * 1.86 \text{ часа} (112 \text{ мин}) = 261.33 \text{ км}$ След увеличението: $140 + 10\% = 154.00 \text{ км/ч} * 1.25 \text{ часа} (75 \text{ мин}) = 192.5 \text{ км}$ След намаляването: $154.00 - 5\% = 146.29 \text{ км/ч} * 3.16 \text{ часа} (190 \text{ мин}) = 463.28 \text{ км}$ Общо изминати: 917.1166 км |

Насоки и подсказки

Вероятно е подобно условие да изглежда на пръв поглед **объркващо** и непълно, което **придава** допълнителна **сложност** на една лесна задача. Нека **разделим** заданието на няколко **подзадачи** и да се опитаме да **решим** всяка една от тях, което ще ни отведе и до крайния резултат:

- Нека **първата** подзадача бъде да **прочетем входните данни**, които потребителя въвежда, и да ги **запазим в подходящи променливи**.
- **Изпълнение** на основната програмна **логика**, което в нашия случай се свежда до прости пресмятания на данните, които вече имаме.
- **Пресмятане** и оформяне на крайния **резултат**.

Съществената част от програмната логика **се изразява** в това да **пресметнем** какво ще бъде **изминатото разстояние** след **всички промени** в скоростта. Тъй като по време на **изпълнението** на програмата, част от **данните**, с които разполагаме, **се променят**, то бихме могли да **разделим** програмния **код** на няколко **логически обособени части**:

- **Пресмятане** на изминатото **разстояние** с първоначална скорост.
- Промяна на **скоростта** и пресмятане на изминатото **разстояние**.

- Последна промяна на **скоростта и пресмятане**.
- **Сумиране**.

За прочитането на данните от конзолата използваме следната **функция**:

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

По условие **входните данни** се въвеждат на **четири** отделни реда, по тази причина следва да изпълним **предходния** код общо **четири** пъти.

```
string initialSpeedString = Console.ReadLine();
// string firstIntervalString = TODO
// string secondIntervalString = TODO
// string thirdIntervalString = TODO
```

За извършване на пресмятанията избираме да използваме тип **decimal**.



Типът данни за реални числа с десетична точност в C# е 128-битовият тип **decimal**. Той има **точност** от **28** до **29** десетични цифри. **Минималната** му стойност е -7.9×10^{28} , а **максималната** е $+7.9 \times 10^{28}$. Стойността му по **подразбиране** е **0.0m** или **0.0M**. Символът '**m**' накрая указва изрично, че числото е от тип **decimal** (по **подразбиране** всички реални числа са от тип **double**). Най-близките до **0** числа, които могат да бъдат записани в **decimal**, са $\pm 1.0 \times 10^{-28}$. Видно е, че **decimal** не може да съхранява **много големи** положителни и отрицателни числа (например със стотици цифри), нито стойности много **близки до 0**. За сметка на това този тип почти **не прави** грешки при **финансови пресмятания**, защото представя числата като **сума от степени на числото 10**, при което **загубите** от закръгления са много **по-малки**, отколкото когато се използва двоично представяне. Реалните числа от тип **decimal** са **изключително удобни за пресмятания с пари** – изчисляване на приходи, задължения, данъци, лихви и т.н.

Повече за различните **типове** данни в езика C# може да прочетете тук:

http://www.introprogramming.info/intro-csharp-book/read-online/glava2-primitivni-tipove-i-promenlivи/#_Toc298863935.

По този начин успяхме да се справим успешно с **първата подзадача**. Следващата стъпка е да преобразуваме **входните данни** в подходящи **типове**, за да можем да извършим необходимите пресмятания. Избираме да използваме тип **Int32** или **int**, тъй като в условието на задачата е упоменато, че входните данни ще бъдат в **определен интервал**, за който този тип данни е напълно достатъчен. Преобразуването извършваме по следния начин:

```
int initialSpeed = int.Parse(initialSpeedString);
// int firstInterval = TODO
// int secondInterval = TODO
// int thirdInterval = TODO
```

Първоначално **запазваме** една **променлива**, която ще използваме многоократно. Този подход на централизация ни дава **Гъвкавост** и **възможност** да **променяме** цялостния резултат на програмата с минимални усилия. В случай, че се наложи да променим стойността, трябва да го направим само на **едно място в кода**, което ни спестява време и усилия.

```
decimal minutesPerHour = 60m;
```



Избягването на повторящ се код (централизация на програмната логика) в задачите, които разглеждаме в настоящата книга, изглежда на пръв поглед излишна, но този подход е от съществено значение при изграждането на мащабни приложения в реална работна среда и упражняването му в начален стадий на изучаване само ще подпомогне усвояването на един качествен стил на програмиране.

Изминалото време (в часове) пресмятаме като **разделим времето на 60** (минутите в един час). **Изминатото разстояние** намираме като **умножим началната скорост с изминалото време** (в часове). След това променяме скоростта, като я увеличаваме с **10%** по условие. Пресмятането на **процентите**, както и следващите изминати **разстояния**, извършваме по следния начин:

- **Интервалът от време** (в часове) намираме като **разделим** зададения интервал в минути на минутите, които се съдържат в един час (60).
- **Изминатото** разстояние намираме като **умножим** интервала (в часове) по скоростта, която получихме след увеличението.
- Следващата стъпка е да **намалим** скоростта с **5%**, както е зададено по условие.
- Намираме оставащото разстояние по описания начин в първите две точки.

```
decimal firstIntervalHours = firstInterval / minutesPerHour;
decimal firstDistance = initialSpeed * firstIntervalHours;
```

```
decimal speedAfterIncrease = initialSpeed
    + ((initialSpeed * 10) / 100m);
decimal secondIntervalHours = secondInterval / minutesPerHour;
decimal secondDistance = speedAfterIncrease * secondIntervalHours;
```

До този момент успяхме да **изпълним** две от **най-важните подзадачи**, а именно **приемането на данните** и **тяхната обработка**. Остава ни само да **пресметнем крайния резултат**. Тъй като по условие се изисква той да бъде **форматиран до 2**

символа след десетичния знак, можем да го направим по следния **начин**:

```
decimal finalDistance =
    firstDistance + secondDistance + thirdDistance;
decimal finalResult = string.Format("{0:f2}", finalResult);
Console.WriteLine(finalResult);
```

В случай че сте работили правилно и изпълните програмата с входните данни от условието на задачата, ще се уверите, че тя работи коректно.

Тестване в Judge системата

Тествайте решението си: <https://judge.softuni.bg/Contests/Practice/Index/517#0>.

Задача: смяна на плочки

Хараламби има събрани пари, с които иска да смени плочките на пода в банята. Като подът е правоъгълник, а плочките са триъгълни. Напишете програма, която да пресмята дали събраните пари ще му стигнат. От конзолата се четат широчината и дължината на пода, както и едната страна на триъгълника с височината към няя. Трябва да пресметнете колко плочки са нужни, за да се покрие пода. Броят на плочките трябва да се закръгли към по-високо цяло число и да се прибавят още 5 броя за фира. От конзолата се четат още – цената на плочка и сумата за работата на майстор.

Входни данни

От конзолата се четат 7 реда:

- Събраните пари.
- Широчината на пода.
- Дължината на пода.
- Страната на триъгълника.
- Височината на триъгълника.
- Цена на една плочка.
- Сумата за майстора.

Всички числа са реални числа в интервала [0.00 ... 5000.00].

Изходни данни

На конзолата трябва да се отпечата на **един ред**:

- Ако парите са достатъчно:
 - “{Оставящите пари} lv left.”

- Ако парите НЕ СА достатъчно:
 - “You'll need {Недостигащите пари} lv more.”

Резултатът трябва да е **форматиран до втория символ** след десетичния знак.

Примерен вход и изход

| Вход | Изход | Обяснения |
|-------|------------------------------------|--|
| 1000 | | Площ на пода → $5.55 * 8.95 = 49.67249$ |
| 5.55 | | Площта на плочка → $0.9 * 0.85 / 2 = 0.3825$ |
| 8.95 | You'll need 1209.65 lv more. | Необходими плочки → $49.67249 / 0.3825 = 129.86\dots = 130 + 5$ фира = 135 |
| 0.90 | | Обща сума → $135 \cdot 13.99 + 321$ (майстор) = 2209.65 |
| 0.85 | | $2209.65 > 1000 \rightarrow$ не достигат 1209.65 лева |
| 13.99 | | |
| 321 | | |

| Вход | Изход | Обяснения |
|------|-------------------|--|
| 500 | | Площ на пода → $3 * 2.5 = 7.5$ |
| 3 | | Площта на плочка → $0.5 * 0.7 / 2 = 0.175$ |
| 2.5 | | Необходими плочки → $7.5 / 0.175 = 42.857\dots = 43 + 5$ |
| 0.5 | | фира = 48 |
| 0.7 | | Обща сума → $48 * 7.8 + 100$ (майстор) = 474.4 |
| 7.80 | | $474.4 < 500 \rightarrow$ остават 25.60 лева |
| 100 | 25.60 lv left. | |

Насоки и подсказки

Следващата задача изисква от нашата програма да приема повече входни данни и извърши по-голям брой изчисления, въпреки че решението е **идентично**. Приемането на данните от потребителя извършваме по добре **познатия ни** начин. Обърнете внимание, че в раздел **Вход** в условието е упоменато, че всички входни данни ще бъдат **реални числа** и поради тази причина бихме използвали тип **decimal**.

След като вече разполагаме с всичко необходимо, за да изпълним програмната логика, можем да пристъпим към следващата част. Как бихме могли да **изчислим** какъв е **необходимият** брой плочки, които ще бъдат достатъчни за покритието на целия под? Условието, че плочките имат **триъгълна** форма, би могло да доведе до объркане, но на практика задачата се свежда до съвсем **прости изчисления**. Бихме могли да пресметнем каква е **общата площ на пода** по формулата за намиране на площ на правоъгълник, както и каква е **площта на една плочка** по съответната формула за триъгълник.

За да пресметнем какъв брой **плочки** са необходими, разделяме площта на пода на **площта на една плочка**(като не забравяме да прибавим 5 допълнителни броя плочки, както е по условие).



Обърнете внимание, че в условието е упоменато да закръглим броя на плочките, получен от делението, до по-високо цяло число, след което да прибавим 5. Потърсете повече информация за системната функционалност за това: **Math.Ceiling(...)**.

До крайния резултат можем да стигнем, като **пресметнем общата сума**, която е необходима, за да бъде покрит целия под, като **съберем цената на плочките с цената за майстора**, която имаме от входните данни.

Можем да се досетим, че **общият разход** за плочките можем да получим, като **умножим броя плочки по цената за плочка**.

Дали сумата, с която разполагаме, ще бъде достатъчна, разбираме като сравним събраните до момента пари (от входните данни) и общите разходи.

Тестване в Judge системата

Тествайте решението си: <https://judge.softuni.bg/Contests/Practice/Index/517#1>.

Задача: магазин за цветя

Магазин за цветя предлага 3 вида цветя: **хризантеми**, **рози** и **лалета**. Цените зависят от сезона.

| Сезон | Хризантеми | Рози | Лалета |
|---------------|--------------|--------------|--------------|
| пролет / лято | 2.00 лв./бр. | 4.10 лв./бр. | 2.50 лв./бр. |
| есен / зима | 3.75 лв./бр. | 4.50 лв./бр. | 4.15 лв./бр. |

В празнични дни цените на всички цветя се **увеличават с 15%**. Предлагат се следните **отстъпки**:

- За закупени повече от 7 лалета през пролетта – **5% от цената на целия букет**.
- За закупени 10 или повече рози през зимата – **10% от цената на целия букет**.
- За закупени повече от 20 цветя общо през всички сезони – **20% от цената на целия букет**.

Отстъпките се правят по така написания ред и могат да се наслагват!

Всички отстъпки важат след осъпяването за празничен ден!

Цената за аранжиране на букета винаги е **2 лв.** Напишете програма, която изчислява **цената за един букет**.

Входни данни

Входът се чете от **конзолата** и съдържа **5 реда**:

- На първи ред е **броят на закупените хризантеми** – цяло число в интервала [0...200].
- На втория ред е **броят на закупените рози** – цяло число в интервала [0...200].
- На третия ред е **броят на закупените лалета** – цяло число в интервала [0...200].
- На четвъртия ред е посочен **сезонът** – [Spring, Summer, Autumn, Winter].
- На петия ред е посочено **дали денят е празник** – [Y – да / N – не].

Изходни данни

Да се отпечата на конзолата 1 число – **цената на цветята**, форматирана до втория символ след десетичния знак.

Примерен вход и изход

| Вход | Изход | Обяснения |
|--------|-------|---|
| 2 | | Цена: $2 * 2.00 + 4 * 4.10 + 8 * 2.50 = 40.40$ лв. |
| 4 | | Празничен ден: $40.40 + 15\% = 46.46$ лв. |
| 8 | 46.14 | 5% намаление за повече от 7 лалета през пролетта: 44.14 |
| Spring | | Общо цветята са 20 или по-малко: няма намаление |
| Y | | $44.14 + 2$ за аранжиране = 46.14 лв. |

| Вход | Изход | Обяснения |
|--------|-------|---|
| 3 | | Цена: $3 * 3.75 + 10 * 4.50 + 9 * 4.15 = 93.60$ лв. |
| 10 | | Не е празничен ден: няма увеличение |
| 9 | 69.39 | 10% намаление за 10 или повече рози през зимата: 84.24 |
| Winter | | Общо цветята са повече от 20: с 20% намаление = 67.392 |
| N | | $67.392 + 2$ за аранжиране = 69.392 лв. |

Насоки и подсказки

След като прочитаме внимателно условието разбираме, че отново се налага да извършваме **прости пресмятания**, но с разликата, че този път ще са необходими и **повече логически проверки**. Следва да обърнем повече **внимание** на това в какъв момент се **извършват промените** по крайната цена, за да можем правилно да изградим логиката на нашата програма. Отново, удебеленият текст ни дава достатъчно **насоки** как да подходим. Като за начало, отделяме вече **декларирани** стойности в **променливи**, както направихме и в предишните задачи:

```
// 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;
```

Правим същото и за останалите вече дефинирани стойности:

```
// 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;
```

Следващата ни подзадача е да прочетем правилно **входните** данни от конзолата. Подхождаме по добре познатия ни вече начин, но този път **комбинираме** две отделни функции – една за **прочитане** на ред от конзолата и друга за **преобразуването** му в числен тип данни:

```
int chrysantemumsPurchased = int.Parse(Console.ReadLine());
// int rosesPurchased = TODO
// int tulipsPurchased = TODO
// string season = TODO
// string isSpecialDay = TODO
```

Нека помислим кой е най-подходящият начин да **структуриме** нашата програма логика. От условието става ясно, че пътят на програмата се разделя основно на две части: **пролет / лято** и **есен / зима**. Разделението правим с условна конструкция, като преди това заделяме променливи за **цените** на отделните цветя, както и за **крайния** резултат.

```
if (season == "Winter" || season == "Autumn")
{
    rosesPrice = rosesPurchases * roseAutumnWinterPrice;
    chrysantemumsPrice = chrysantemumsPurchases
        * chrysantemumAutumnWinterPrice;
```

```

    tulipsPrice = tulipsPurchased * tulipAutumnWinterPrice;
    totalCost = rosesPrice + chrysantemumsPrice + tulipsPrice;
}
else
{
    // TODO
}

```

Остава ни да извършим **няколко проверки** относно **намаленията** на различните видове цветя, в зависимост от сезона, и да модифицираме крайния резултат.

Тестване в Judge системата

Тествайте решението си: <https://judge.softuni.bg/Contests/Practice/Index/517#2>.

Задача: оценки

Напишете програма, която да **пресмята статистика на оценки** от изпит. В началото програмата получава **броя на студентите**, явили се на изпита и за **всеки студент неговата оценка**. На края програмата трябва да **отпечата процента на студенти** с оценка между 2.00 и 2.99, между 3.00 и 3.99, между 4.00 и 4.99, 5.00 или повече, както и **средният успех** на изпита.

Входни данни

От конзолата се четат **поредица** от числа, всяко на отделен ред:

- На първия ред – **броя на студентите явили се на изпит** – цяло число в интервала [1...1000].
- За **всеки един студент** на отделен ред – **оценката от изпита** – реално число в интервала [2.00...6.00].

Изходни данни

Да се отпечатат на конзолата **5 реда**, които съдържат следната информация:

- "Top students: {процент студенти с успех 5.00 или повече}%".
- "Between 4.00 and 4.99: {между 4.00 и 4.99 включително}%".
- "Between 3.00 and 3.99: {между 3.00 и 3.99 включително}%".
- "Fail: {по-малко от 3.00}%".
- "Average: {среден успех}%".

Примерен вход и изход

| Вход | Изход | Обяснения |
|------|-------------------------------|---|
| 10 | | |
| 3.00 | | |
| 2.99 | | 5 и повече – трима = 30% от 10 |
| 5.68 | Top students: 30.00% | Между 4.00 и 4.99 – трима = 30% от 10 |
| 3.01 | Between 4.00 and 4.99: 30.00% | Между 3.00 и 3.99 – двама = 20% от 10 |
| 4 | Between 3.00 and 3.99: 20.00% | Под 3 – двама = 20% от 10 |
| 4 | Fail: 20.00% | Средният успех е: $3 + 2.99 + 5.68 + 3.01 + 4 + 4 + 6 + 4.50 + 2.44 + 5 = 40.62 / 10 = 4.062$ |
| 6.00 | Average: 4.06 | |
| 4.50 | | |
| 2.44 | | |
| 5 | | |

| Вход | Изход |
|------|-------------------------------|
| 6 | |
| 2 | Top students: 33.33% |
| 3 | Between 4.00 and 4.99: 16.67% |
| 4 | Between 3.00 and 3.99: 16.67% |
| 5 | Fail: 33.33% |
| 6 | Average: 3.70 |
| 2.2 | |

Насоки и подсказки

От условието виждаме, че **първо** ще ни бъде подаден **броя** на студентите, а едва **след това оценките** им. По тази причина **първо** в една променлива от тип **int** ще прочетем **броя** на студентите. За да прочетем и обработим самите оценки, ще използваме **for** цикъл. Стойността на променливата **int** ще бъде **краината** стойност на променливата **i** от цикъла. По този начин **всички** итерации на цикъла ще прочетат **всяка една оценка**.

```
int numberOfStudents = int.Parse(Console.ReadLine());
for (int i = 0; i < numberOfStudents; i++)
{
    // TODO
}
```

Преди да се изпълни кода от **for** цикъла заделяме променливи, в които ще пазим **броя на студентите** за всяка група: слаби резултати (до 2.99), резултати от 3 до 3.99, от 4 до 4.99 и оценки над 5. Ще ни е необходима и още една променлива, в която да пазим **сумата на всички оценки**, с помощта на която ще изчислим средната оценка на всички студенти.

```
double numberOfFailedStudents = 0;
double numberOfAverageStudents = 0;
double numberOfGoodStudents = 0;
double numberOfExcellentStudents = 0;
double totalResult = 0;
```

Завъртаме цикъла и в него **декларираме още една** променлива, в която ще запазваме **текущата** въведена оценка. Променливата ще е от тип **double** и на всяка итерация проверяваме **каква е стойността** ѝ. Според тази стойност, **увеличаваме** броя на студентите в съответната група с **1**, като не забравяме да увеличим и **общата** сума на оценките, която също следим.

```
for (int i = 0; i < numberOfStudents; i++)
{
    double grade = double.Parse(Console.ReadLine());
    totalResult += grade;
    if (grade < 3)
    {
        numberOfFailedStudents++;
    }
    else // TODO: check other groups
}
```

Какъв **процент** заема дадена група **студенти** от общия брой, можем да пресметнем като **умножим** броя на **студентите** от съответната група по **100** и след това **разделим** на **общия брой студенти**.



Обърнете внимание с какъв числен тип данни работите при извършване на тези пресмятания.

Крайният резултат оформяме по добре познатия ни начин **до втория символ** след десетичния знак.

Тестване в Judge системата

Тествайте решението си: <https://judge.softuni.bg/Contests/Practice/Index/517#3>.

Задача: коледна шапка

Да се напише програма, която прочита от конзолата **цяло число n** и чертае **коледна шапка** с ширина $4 * n + 1$ колони и височина $2 * n + 5$ реда като в примерите подолу.

Входни данни

Входът се чете от конзолата – едно **цяло число n** в интервала [3...100].

Изходни данни

Да се отпечата на конзолата **коледна шапка**, точно както в примерите.

Примерен вход и изход

Насоки и подсказки

При задачите за **чертане** с конзолата, най-често потребителят въвежда **едно цяло число**, което е свързано с **общата големина на фигурката**, която трябва да начертаем. Тъй като в условието е упоменато как се изчисляват общата дължина и широчина на фигурката, можем да ги използваме за **отправни точки**. От примерите ясно се вижда, че без значение какви са входните данни, винаги имаме **първи два реда**, които са с почти идентично съдържание.

..... / \

Забелязваме също така, че **последните три реда** винаги присъстват, **два** от които са напълно **еднакви**.

```
*****
*.*.*.*.*.*.*
*****
```

От тези наши наблюдения можем да изведем **формулата за височина на променливата част** на коледната шапка. Използваме зададената по условие формула за общата височина, като изваждаме големината на непроменливата част. Получаваме **(2 * n + 5) - 5** или **2 * n**.

За начертаването на **динамичната** или променлива част от фигурката ще използваме **цикъл**. Размерът на цикъла ще бъде от **0** до **широцната**, която имаме по условие, а именно **4 * n + 1**. Тъй като тази формула ще използваме на **няколко места** в кода, е добра практика да я изнесем в **отделна променлива**. Преди изпълнението на цикъла би следвало да **заделим променливи** за броя на отделните символи, които участват в динамичната част: **точки и тирета**. Чрез изучаване на примерите можем да изведем формули и за **стартовите стойности** на тези променливи. Първоначално **ти retata** са **0**, но броя на **точките** ясно се вижда, че можем да получим като от **общата широчина** извадим **3** (броя символи, които изграждат върха на коледната шапка) и след това **разделим на 2**, тъй като броя точки от двете страни на шапката е еднакъв.

```
***.
....*-*-*.
....*--*--*.
....*---*---*.
...*----*----*.
..*-----*--*..
.*-----*-----*.
*-----*-----*
```

Остава да изпълним тялото на цикъла, като **след всяко** начертаване **намалим** броя на точки с **1**, а **ти retata увеличим** с **1**. Нека не забравяме да начертаем и по една **звездичка** между тях. Последователността на чертане в тялото на цикъла е следната:

- Символен низ от точки
- Звезда
- Символен низ от тирета
- Звезда
- Символен низ от тирета
- Звезда
- Символен низ от точки

В случай че сме работили правилно получаваме фигурки, идентични на тези от примерите.

Тестване в Judge системата

Тествайте решението си: <https://judge.softuni.bg/Contests/Practice/Index/517#4>.

Задача: комбинации от букви

Напишете програма, която да принтира на конзолата **всички комбинации от 3 букви** в зададен интервал, като се пропускат комбинациите, **съдържащи** зададена от конзолата **буква**. Накрая трябва да се принтира броят отпечатани комбинации.

Входни данни

Входът се чете от **конзолата** и съдържа **точно 3 реда**:

- Малка буква от английската азбука за **начало** на интервала – от 'a' до 'z'.
- Малка английска буква за **край** на интервала – от **първата буква** до 'z'.
- Малка буква от английската азбука – от 'a' до 'z' – като комбинациите, които съдържат тази буква **се пропускат**.

Изходни данни

Да се отпечатат на един ред **всички комбинации**, отговарящи на условието, следвани от **броя им**, разделени с интервал.

Примерен вход и изход

| Вход | Изход | Обяснения |
|-------------|--------------------------------------|---|
| a c b | aaa aac aca acc caa cac cca ccc 8 | Всички възможни комбинации с буквите 'a', 'b' и 'c' са: 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 Комбинациите, съдържащи 'b' , не са валидни. Остават 8 валидни комбинации. |

| Вход | Изход |
|-------------|---|
| f k h | fff ffg ffi ffj ffk fgf fgg fgi fgj fgk fif fig fii fij fik fjf fji fjj fjk fkf fkg fki fkj fkk gff gfg gfi gfj gfk ggf ggg ggi ggi ggk gif gig gii gjj gik gif gjg gjj gjj gjk gkf gkg gki gkj gkk iff ifg ifi ifj ifk ifg iff iif iif iig iii iij iik iif iig ijj iij ijk ikf ikg iki ikj ijk ijk jff jfg jfi jfj jfk jgf jgg jgi jgj jgk jif jig jii jik jjf jgg jji jjj jjk jkf jkg jki jkj jkk kff kfg kfi kfj kfk kgf kgg kgi kgj kgk kif kig kii kij kik kfj kkg kj kjj kjk kkf kkg kki kkj kkk 125 |

| Вход | Изход |
|-------------|---|
| a c z | 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 27 |

Насоки и подсказки

За последната задача имаме по условие входни данни на **3 реда**, които са представени от по един символ от **ASCII таблицата** (<http://www.asciitable.com>). Бихме могли да използваме вече **дефинирана функция** в езика C#, като преобразуваме входните данни в тип данни **char** по следния начин:

```
char startLetter = char.Parse(Console.ReadLine());
// TODO
```

Нека помислим как бихме могли да стигнем до **крайния резултат**. В случай че условието на задачата е да се принтират всички от началния до крайния символ (с пропускане на определена буква), как бихме постъпили?

Най-лесният и удачен начин е да използваме **цикъл**, като преминем през **всички символи** и принтираме тези, които са **различни от буквата**, която трябва да пропуснем. Едно от предимствата на езика C#, е че имаме възможност да използваме различен тип данни за циклична променлива. В примерния код по-долу правим цикъл от буквата **"a"** до буквата **"z"** включително, като преминаваме последователно през всички малки букви от латинската азбука:

```
for (char i = 'a'; i <= 'z'; i++)
{
    Console.WriteLine(i + " ");
}
```

Резултатът от изпълнението на кода е всички букви от **a** до **z** включително, принтирани на един ред и разделени с интервал. Това прилика ли на крайния резултат от нашата задача? Трябва да измислим **начин**, по който да се принтират по **3 символа**, както е по условие, вместо по **1**. Изпълнението на програмата много прилича на игрална машина, в която печелим, ако успеем да наредим няколко еднакви символа. Да речем, че на машината имаме места за три символа. Когато **спрем** на даден **символ** на първото място, на останалите две места **продължават** да се изреждат символи от всички възможни. В нашия случай **всички възможни** са буквите от началната до крайната такава, зададена от потребителя, а решението на нашата програма е идентично на начина, по който работи игралната машина.

Използваме **цикъл**, който минава през **всички символи** от началната до крайната буква включително. На **всяка итерация** на **първия цикъл** пускаме **втори** със същите параметри (но **само ако** буквата на първия цикъл е валидна, т.е. не съвпада с тази, която трябва да изключим по условие). На всяка итерация на **втория цикъл**

пускаме още **един** със **същите параметри** и **същата проверка**. По този начин имаме три вложени цикъла, като в тялото на **последния** принтираме символите.

```
for (char i = startLetter; i <= endLetter; i++)
{
    if (i != exceptLetter)
    {
        // TODO
    }
}
```

Нека не забравяме, че се изиска от нас да принтираме и **общия брой валидни комбинации**, които сме намерили, както и че те трябва да се принтират на **същия ред**, разделени с интервал.

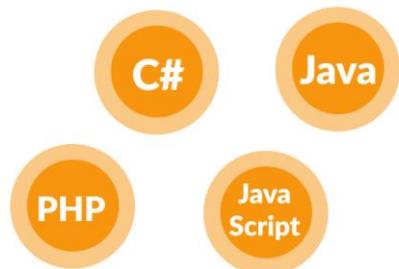
Тестване в Judge системата

Тествайте решението си: <https://judge.softuni.bg/Contests/Practice/Index/517#5>.

Качествено образование,
професия и работа за

Софтуерни инженери

- ✓ Безплатен старт за начинаещи - **присъствено и онлайн**
- ✓ Избор измежду **най-търсените професии** в софтуерната индустрия
- ✓ Съдействие за **кариерен старт**
- ✓ Отлични преподаватели, ментори и студентска общност



Софтууни предоставя съвременно иновативно образование за програмиране, ИТ и дигитални умения на хиляди млади хора. Програмата "**Софтуерен университет**" изгражда истински **професионалисти в света на програмирането**.

Включете се в цялостната програма по софтуерно инженерство на Софтууни, за да усвоите **най-търсените съвременни софтуерни технологии** чрез модерна учебна методика и с много практически задачи и проекти. **Учебният план** е разработен съвместно с ИТ фирмите и изгражда най-търсените от тях умения.

Софтууни работи пряко с компаниите от софтуерната индустрия, съдейтайки на своите студенти за реализацията им като успешни софтуерни инженери.

ПЪТЯТ НА СТУДЕНТА В СОФТУНИ



Programming Basics

Кариерен Старт

Дипломиране

70/100 credits

80/100/150 credits

Кандидатствай

softuni.bg/apply

Глава 9.1. Задачи за шампиони – част I

В настоящата глава ще предложим на читателя няколко малко по-трудни задачи, които имат за цел развиване на алгоритмични умения и усвояване на програмни техники за решаване на задачи с по-висока сложност.

По-сложни задачи върху изучавания материал

Ще решим заедно няколко задачи по програмиране, които обхващат изучавания в книгата учебен материал, но по трудност надвишават обичайните задачи от приемните изпити в СофтУни. Ако искате да станете шампиони по основи на програмирането, ви препоръчваме да тренирате решаване на подобни по-сложни задачи, за да ви е лесно на изпитите.

Задача: пресичащи се редици

Имаме две редици:

- **редица на Трибоначи** (по аналогия с редицата на Фиbonacci), където всяко число е **сумата от предните три** (при дадени начални три числа)
- редица, породена от **числова спирала**, дефинирана чрез обхождане като **спирала** (дясно, долу, ляво, горе, дясно, долу, ляво, горе и т.н.) на матрица от числа, стартирайки от нейния център с дадено начално число и стъпка на увеличение, със записване на текущите числа всеки път, когато направим завой.

Да се напише програма, която намира първото число, което се появява **и в двете** така дефинирани редици.

Пример

Нека **редицата на Трибоначи** да започне с 1, 2 и 3. Това означава, **първата редица** че ще съдържа числата 1, 2, 3, 6, 11, 20, 37, 68, 125, 230, 423, 778, 1431, 2632, 4841, 8904, 16377, 30122, 55403, 101902 и т.н.

Същевременно, нека **числата в спиралата** да започнат с 5 и спиралата да се увеличава с 2 на всяка стъпка. Тогава **втората редица** ще съдържа числата 5, 7, 9, 13, 17, 23, 29, 37 и т.н. Виждаме, че 37 е първото число, което се среща в редицата на Трибоначи и в спиралата и това е търсеното решение на задачата.

Входни данни

Входните данни трябва да бъдат прочетени от конзолата.

| | | | | | |
|-----|-----|-----|-----|-----|-----|
| 45 | ... | ... | ... | ... | 55 |
| ... | 17 | 19 | 21 | 23 | ... |
| ... | 15 | 5 | 7 | ... | ... |
| ... | 13 | 11 | 9 | ... | ... |
| 37 | ... | ... | ... | 29 | ... |
| ... | ... | ... | ... | ... | 65 |

- На първите три реда от входа, ще прочетете **три цели числа**, представляващи **първите три числа** в редицата на Трибоначи.

- На следващите два реда от входа, ще прочетете **две цели числа**, представляващи **първото число и стъпката** за всяка клетка на матрицата за спиралата от числа.

Входните данни винаги ще бъдат валидни и винаги ще са в описания формат. Няма нужда да ги проверявате.

Изходни данни

Резултатът трябва да бъде принтиран на конзолата.

На единствения ред от изхода трябва да принтирате **най-малкото число**, което се среща и в двете последователности. Ако няма число в диапазона [1...1 000 000], което да се среща и в двете последователности, принтирайте "No".

Ограничения

- Всички числа във входа ще бъдат в диапазона [1...1 000 000].
- Позволено работно време за програмата: 0.25 секунди.
- Позволена памет: 16 MB.

Примерен вход и изход

| Вход | Изход | Вход | Изход | Вход | Изход | Вход | Изход |
|------|-------|------|-------|------|-------|------|-------|
| 1 | | 13 | | 99 | | 1 | |
| 2 | | 25 | | 99 | | 4 | |
| 3 | 37 | 99 | 13 | 99 | No | 7 | |
| 5 | | 5 | | 2 | | 23 | |
| 2 | | 2 | | 2 | | 3 | |

Насоки и подсказки

Задачата изглежда доста сложна и затова ще я разбием на по-прости подзадачи.

Обработване на входа

Първата стъпка от решаването на задачата е да прочетем и обработим входа. Входните данни се състоят от **5 цели числа**: **3** за редицата на Трибоначи и **2** за числовата спирала.

```
int tribonacciFirst = int.Parse(Console.ReadLine());
//TODO: Read remaining numbers
```

След като имаме входните данни, трябва да помислим как ще генерираме числата в двете редици.

Генериране на редица на Трибоначи

За редицата на Трибоначи всеки път ще събираме предишните три стойности и след това ще отнемваме стойностите на тези числа (трите предходни) с една позиция напред в редицата, т.е. стойността на първото трябва да приеме стойността на второто и т.н. Когато сме готови с числата, ще запазваме стойността му в **масив**. Понеже в условието на задачата е казано, че числата в редиците не превишават 1,000,000, можем да спрем генерирането на тази редица именно при 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;
}
```

Генериране на числова спирала

Трябва да измислим **зависимост** между числата в числовата спирала, за да можем лесно да генерираме всяко следващо число, без да се налага да разглеждаме матрици и тяхното обхождане. Ако разгледаме внимателно картиката от условието, можем да забележим, че **на всеки 2 "завоя"** в спиралата числата, които прескачаме, се увеличават **с 1**, т.е. от 5 до 7 и от 7 до 9 не се прескача нито 1 число, а директно **събираме със стъпката** на редицата. От 9 до 13 и от 13 до 17 прескачаме едно число, т.е. събираме два пъти стъпката. От 17 до 23 и от 23 до 29 прескачаме две числа, т.е. събираме три пъти стъпката и т.н.

Така виждаме, че при първите две имаме **последното число + 1 * стъпката**, при следващите две събираме **с 2 * стъпката** и т.н. Всеки път, когато искаме да стигнем до следващото число от спиралата, ще трябва да извършваме такива изчисления:

```
spiralCurrent += spiralStep * spiralStepMul;
```

Това, за което трябва да се погрижим, е **на всеки две числа нашият множител** (нека го наречем "кофициент") **да се увеличава с 1** (**spiralStepMul++**), което може да се постигне с прости проверки (**spiralCount % 2 == 0**). Целият код от генерирането на спиралата в **масив** е даден по-долу.

```
var spiralNumbers = new List<int>() { spiralCurrent };
var spiralCount = 0;
var spiralStepMul = 1;

while (spiralCurrent < 1000000)
{
    spiralCurrent += [REDACTED]

    spiralNumbers.Add(spiralCurrent);
    spiralCount++;

    if (spiralCount % 2 == 0)
    {
        spiralStepMul++;
    }
}
```

Намиране на общо число за двете редици

След като сме генерирали числата и в двете редици, можем да пристъпим към обединението им и изграждането на крайното решение. Как ще изглежда то? За **всяко от числата** в едната редица (започвайки от по-малкото) ще проверяваме дали то съществува в другата. Първото число, което отговаря на този критерий ще бъде **отговорът** на задачата.

Търсенето във втория масив ще направим **линейно**, а за по-любопитните ще оставим да си го оптимизират, използвайки техниката наречена **двоично търсене**, тъй като вторият масив се генерира сортиран, т.е. отговаря на изискването за прилагането на този тип търсене. Кодът на нашето решение ще изглежда така:

```
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)
        {
            found = true;
            break;
        }
    }
}

if (!found)
{
    Console.WriteLine("No such number exists!");
}
```

```

        Console.WriteLine(tribonacciNumbers[i]);
        found = true;
        break;
    }
}

if (found)
{
    break;
}
}

if (!found)
{
    Console.WriteLine("No");
}

```

Решението на задачата използва масиви за запазване на стойностите. Масивите не са необходими за решаването на задачата. Съществува **алтернативно решение**, което генерира числата и работи директно с тях, вместо да ги записва в масив. Можем на **всяка стъпка** да проверяваме дали **числата от двете редици съвпадат**. Ако това е така, ще принтираме на конзолата числото и ще прекратим изпълнението на нашата програма. В противен случай, ще видим текущото число на **коя редица** е по-малко и ще генерираме следващото, там където "изоставаме". Идеята е, че **ще генерираме числа от редицата, която е "по-назад"**, докато не прескочим текущото число на другата редица и след това обратното, а ако междувременно намерим съвпадение, ще прекратим изпълнението.

```

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
    }
}

```

Тестване в Judge системата

Тествайте решението си: <https://judge.softuni.bg/Contests/Practice/Index/518#0>.

Задача: магически дати

Да приемем, че ни е зададена **дата** във формат "**дд-мм-гггг**", напр. 17-03-2007. Изчисляваме **теглото на тази дата**, като вземем всичките ѝ цифри, умножим всяка цифра с останалите и накрая съберем всички получени резултати. В нашия случай имаме 8 цифри: **17032007**, така че теглото е $1^*7 + 1^*0 + 1^*3 + 1^*2 + 1^*0 + 1^*0 + 1^*7 + 7^*0 + 7^*3 + 7^*2 + 7^*0 + 7^*0 + 7^*7 + 0^*3 + 0^*2 + 0^*0 + 0^*0 + 0^*7 + 3^*2 + 3^*0 + 3^*0 + 3^*7 + 2^*0 + 2^*7 + 0^*0 + 0^*7 + 0^*7 = 144$.

Нашата задача е да напишем програма, която намира всички **магически дати** – дати между две определени години, отговарящи на дадено магическо тегло. Датите трябва да бъдат принтирани в нарастващ ред във формат "**дд-мм-гггг**". Ще използваме традиционен календар (годините имат 12 месеца, всеки месец има 28, 29, 30 или 31 дни).

Входни данни

Входните данни трябва да бъдат прочетени от конзолата. Състоят се от 3 реда:

- Първият ред съдържа цяло число: **начална година**.
- Вторият ред съдържа цяло число: **краяна година**.
- Третият ред съдържа цяло число: **магическо тегло**.

Входните данни винаги ще бъдат валидни и винаги ще са в описания формат. Няма нужда да ги проверяваме.

Изходни данни

Резултатът трябва да бъде принтиран на конзолата, като последователни дати във формат "**дд-мм-гггг**", подредени по азбучен ред. Всеки низ трябва да е на отделен ред. В случай, че няма съществуващи магически дати, принтираме "**No**".

Ограничения

- Началната и крайната година са цели числа в периода [1900-2100].
- Магическото тегло е цяло число в диапазона [1...1000].
- Позволено работно време за програмата: 0.25 секунди.
- Позволена памет: 16 MB.

Примерен вход и изход

| Вход | Изход | Вход | Изход |
|------|------------|------|------------|
| 2007 | 17-03-2007 | | 09-01-2013 |
| 2007 | 13-07-2007 | | 17-01-2013 |
| 144 | 31-07-2007 | | 23-03-2013 |
| Вход | Изход | | 11-07-2013 |
| 2003 | | | 01-09-2013 |
| 2004 | No | | 10-09-2013 |
| 1500 | | | 09-10-2013 |
| Вход | Изход | | 17-10-2013 |
| 2011 | 01-01-2011 | | 07-11-2013 |
| 2012 | 10-01-2011 | | 24-11-2013 |
| 14 | 01-10-2011 | | 14-12-2013 |
| | 10-10-2011 | | 23-11-2014 |

Насоки и подсказки

Започваме от входните данни. В случая имаме **3 цели числа**, които трябва да се прочетат от конзолата, като с това се изчерпва въвеждането и обработването на входа за задачата.

Разполагайки с началната и крайната година, е хубаво да разберем как ще минем през всяка дата, без да се объркваме от това колко дена има в месеца и дали е високосна година и т.н.

Обхождане на всички дати

За обхождането ще се възползваме от функционалността на класа **DateTime**. Ще си дефинираме **променлива за началната дата**, което можем да направим, използвайки конструктора, който приема година, месец и ден. Знаем, че годината е началната година, която сме прочели от конзолата, а месеца и деня трябва да са съответно януари и 1-ви.

```
DateTime currentDate = new DateTime(startYear, 1, 1);
```

След като имаме началната дата, искаме да направим **цикъл**, който се изпълнява, докато не превишим **крайната година** (или докато не преминем 31 декември в крайната година, ако сравняваме целите дати), като на всяка стъпка увеличава с по 1 ден.

За да увеличаваме с 1 ден при всяко завъртане, ще използваме метода **AddDays(...)** от класа **DateTime**, който добавя брой дни към текущата дата и връща новополучената.

Внимание: тъй като методът **DateTime.AddDays(...)** връща "новата" дата, е важно да имаме присвояване на резултата, а не само извикване на метода! Методът ще

се грижи вместо нас кога трябва да прескочи в следващия месец, колко дни има даден месец и всичко около високосните години.

В крайна сметка нашият цикъл може да изглежда по следния начин:

```
while (currentDate.Year <= endYear)
{
    //TODO: Do all the magic

    currentDate = currentDate.AddDays(1);
}
```

Забележка: може да постигнем същия резултат с **for** цикъл, инициализацията на датата отива в първата част на **for**, условието се запазва, а стъпката е увеличаването с 1 ден. Също и условието може да се замени като се направи **променлива за крайната дата**, т.е. 31 декември в крайната година и да се сравняват директно двете дати.

Пресмятане на теглото

Всяка дата се състои от точно 8 символа (цифри): 2 за **дения (d1, d2)**, 2 за **месеца (d3, d4)** и 4 за **годината (d5 до d8)**. Това означава, че всеки път ще имаме едно и също пресмятане и може да се възползваме от това, за **да дефинираме формулата статично** (т.е. да не обикаляме с цикли, реферирайки различни цифри от датата, а да изпишем цялата формула). За да успеем да я изпишем, ще ни трябват **всички цифри от датата** в отделни променливи, за да направим всички нужни умножения. Използвайки операциите деление и взимане на остатък върху отделните компоненти на датата, чрез свойствата **Day, Month** и **Year**, можем да извлечем всяка цифра.

```
int d1 = currentDate.Day / 10; // First day digit
int d2 = currentDate.Day % 10; // Second day digit

int d3 = currentDate.Month / 10; // First month digit
int d4 = currentDate.Month % 10; // Second month digit

int d5 = currentDate.Year / 1000; // First year digit
int d6 = (currentDate.Year / 100) % 10; // Second year digit
int d7 = currentDate.Year % 100; // Third year digit
int d8 = currentDate.Year % 10; // Fourth year digit
```

Нека обясним и един от по-интересните редове тук. Нека вземе за пример взимането на втората цифра от годината (**d6**). При нея делим годината на 100 и взимаме остатък от 10. Какво постигаме така? Първо с деленето на 100 отстраняваме последните 2 цифри от годината (пример: $2018 / 100 = 20$). С

остатъка от деление на 10 взимаме последната цифра на полученото число (**20 % 10 = 0**) и така получаваме 0, което е втората цифра на 2018.

Остава да направим изчислението, което ще ни даде магическото тегло на дадена дата. За да **не изписваме всички умножения**, както е показано в примера, ще приложим просто групиране. Това, което трябва да направим, е да умножим всяка цифра с тези, които са след нея. Вместо да изписваме $d1 * d2 + d1 * d3 + \dots + d1 * d8$, може да съкратим този израз до $d1 * (d2 + d3 + \dots + d8)$, следвайки математическите правила за групиране, когато имаме умножение и събиране. Прилагайки същото опростяване за останалите умножения, получаваме следната формула:

```
dateWeight = d1 * (d2 + d3 + d4 + d5 + d6 + d7 + d8) +
    // d2 * ... +
    //...
```

Отпечатване на изхода

След като имаме пресметнатото теглото на дадена дата, трябва **да проверим дали съвпада с търсеното от нас магическо тегло**, за да знаем, дали трябва да се принтира или не. Проверката може да се направи със стандартен **if** блок, като трябва да се внимава при принтирането датата да е в правилния формат.

```
if (dataWeight == numberToSearchFor)
{
    // Print
    found = true;
}
```

За отпечатването на датите имаме два варианта:

- Първият начин е да използваме метода **.ToString(...)**, на който можем да **подадем формат на датата**, т.е. дали дните да се изписват с водеща нула или не, дали месеците да се изписват с водещи нули или не, с думи или с цифри, с кратък запис или с пълно име и т.н.

```
Console.WriteLine(currentDate.ToString("Date Format Goes Here"));
```

- Вторият вариант е да вземем отделните компоненти на датата **Day**, **Month** и **Year**, както направихме при пресмятането, и да си оформим изхода чрез **форматиращ стринг**.

```
Console.WriteLine($"{0}-{1}-{2}", /*Date components goes here*/);
```

Внимание: тъй като обхождаме датите от началната година към крайната, те винаги ще бъдат подредени във възходящ ред, както е по условие.

И накрая, ако не сме намерили нито една дата, отговаряща на условията, ще имаме **false** стойност във **found** променливата и ще можем да отпечатаме **No**.

```
if (!found)
{
    Console.WriteLine("No");
}
```

Тестване в Judge системата

Тествайте решението си: <https://judge.softuni.bg/Contests/Practice/Index/518#1>.

Задача: пет специални букви

Дадени са две числа: **начало** и **край**. Напишете програма, която генерира всички комбинации от 5 букви, всяка измежду множеството {'a', 'b', 'c', 'd', 'e'}, така че теглото на тези 5 букви да е число в интервала [начало...край] включително. Принтирайте ги по азбучен ред, на един ред, разделени с интервал.

Теглото на една буква се изчислява по следния начин:

```
weight('a') = 5; weight('b') = -12; weight('c') = 47;
weight('d') = 7; weight('e') = -32;
```

Теглото на редицата от букви $c_1 c_2 \dots c_n$ се изчислява, като се премахват всички букви, които се повтарят (от дясно наляво), и след това се пресметне формулата:

```
weight(c1c2...cn) = 1 * weight(c1) + 2 * weight(c2) + ... + n *
weight(cn)
```

Например, теглото на "bcddc" се изчислява по следния начин:

Първо премахваме повторящите се букви и получаваме "bcd". След това прилагаме формулата:

$$1 * \text{weight}('b') + 2 * \text{weight}('c') + 3 * \text{weight}('d') = 1 * (-12) + 2 * 47 + 3 * 7 = 103.$$

Друг пример: $\text{weight}("cadea") = \text{weight}("cade") = 1 * 47 + 2 * 5 + 3 * 7 - 4 * 32 = -50$.

Входни данни

Входните данни се четат от конзолата. Състоят се от два реда:

- Числото за **начало** е на първия ред.
- Числото за **край** е на втория ред.

Входните данни винаги ще бъдат валидни и винаги ще са в описания формат. Няма нужда да ги проверявате.

Изходни данни

Резултатът трябва да бъде принтиран на конзолата като поредица от низове, подредени по азбучен ред. Всеки низ трябва да бъде отделен от следващия с едно разстояние. Ако теглото на нито един от 5 буквните низове не съществува в зададения интервал, принтирайте "No".

Ограничения

- Числата за начало и край да бъдат цели в диапазона [-10000...10000].
- Позволено работно време за програмата: 0.25 секунди.
- Позволена памет: 16 MB.

Примерен вход и изход

| Вход | Изход | Обяснения | Вход | Изход |
|------|---|----------------------|------|-------|
| 40 | bcead | weight("bcead") = 41 | 300 | |
| 42 | bdcea | weight("bdcea") = 40 | 400 | No |
| Вход | Изход | | | |
| -1 | bcdea cebda eaaad eaada eaadd eaade eaaed eadaa eadad eadae eadda eaddd eadde eadea eaded eadee eaead eaeda eaedd eaede eaeed eeaad eeada eeadd eeade eeaed eeead | | | |
| 1 | | | | |
| 200 | baadc babdc badac badbc badca badcb badcc badcd baddc bbadc bbdac bdaac bdabc bdaca bdacb bdacc bdacd bdadc bdbac bddac beadc bedac eabdc ebadc edbac edbac | | | |
| 300 | | | | |

Насоки и подсказки

Като всяка задача, започваме решението с прочитане и обработване на входните данни. В случая имаме две цели числа, които можем да обработим с комбинация от методите `int.Parse(...)` и `Console.ReadLine()`.

```
int firstNumber = int.Parse(Console.ReadLine());
int secondNumber = int.Parse(Console.ReadLine());
```

В задачата имаме няколко основни момента - генерирането на всички комбинации с дължина 5 включващи 5-те дадени букви, премахването на повтарящите се букви и пресмятането на теглото за дадена вече опростена дума. Отговорът ще се състои от всяка дума, чието тегло е в дадения интервал [`firstNumber`, `secondNumber`].

Генериране на всички комбинации

За да генерираме всички комбинации с дължина 1 използвайки 5 символа, бихме използвали цикъл от 0..4, като всяко число от цикъла ще искаем да отговаря на

един символ. За да генерираме всички комбинации с дължина 2 използвайки 5 символа (т.e. "aa", "ab", "ac", ..., "ba", ...), бихме направили два вложени цикъла, всеки обхождащ цифрите от 0 до 4, като отново ще направим, така че всяка цифра да отговаря на конкретен символ. Тази стъпка ще повторим 5 пъти, така че накрая да имаме 5 вложени цикъла с индекси **i1**, **i2**, **i3**, **i4** и **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++)
                {
                }
            }
        }
    }
}
```

Имайки всички 5-цифрени комбинации, трябва да намерим начин да "превърнем" петте цифри в дума с буквите от 'a' до 'e'. Един от начините да направим това е, като си **предефинираме** прост стринг съдържащ буквите, които имаме

```
string pattern = "abcde";
```

и за всяка цифра взимаме буквата от конкретната позиция. По този начин числото 00000 ще стане "aaaaa", числото 02423 ще стане "acecd". Можем да направим стринга от 5 букви по следния начин.

```
string fullWord = "" + pattern[i1]
                  + pattern[i2]
                  + pattern[i3]
                  + pattern[i4]
                  + pattern[i5];
```

Друг начин: можем да преобразуваме цифрите до букви, използвайки подредбата им в ASCII таблицата. Изразът '**a**' + **i** ще ни даде резултата '**a**' при **i** = 0, '**b**' при **i** = 1, '**c**' при **i** = 2 и т.н.

Така вече имаме генериирани всички 5-буквени комбинации и можем да продължим със следващата част от задачата.

Внимание: тъй като сме подбрали **pattern**, съобразен с азбучната подредба на буквите и циклите се въртят по подходящ начин, алгоритъмът ще генерира думите в азбучен ред и няма нужда от допълнително сортиране преди извеждане.

Премахването на повтарящи се букви

След като имаме вече готовия низ, трябва да премахнем всички повтарящи се символи. Ще направим тази операция, като **добавяме буквите от ляво надясно в нов низ и всеки път преди да добавим буква ще проверяваме дали вече я има** - ако я има ще я пропускаме, а ако я няма ще я добавяме. За начало ще добавим първата буква към началния стринг.

```
string word = pattern[i1].ToString();
```

След това ще направим същото и с останалите 4, проверявайки всеки път дали ги има със следното условие и метода **.IndexOf(...)**. Това може да стане с цикъл по **fullWord** (оставяме това на читателя за упражнение), а може да стане и по мързеливия начин с copy-paste.

```
if (word.IndexOf(pattern[i2]) == -1)
    word += pattern[i2];
// ...
```

Методът **.IndexOf(...)** връща индекса на конкретния елемент, ако бъде намерен или **-1**, ако елементът не бъде намерен. Следователно всеки път, когато получим **-1**, ще означава, че все още нямаме тази буква в новия низ с уникални букви и можем да я добавим, а ако получим стойност различна от **-1**, ще означава, че вече имаме буквата и няма да я добавяме.

Пресмятане на теглото

Пресмятането на теглото е просто **обхождане на уникалната дума (word)**, получена в миналата стъпка, като за всяка буква трябва да вземем теглото ѝ и да я умножим по позицията. За всяка буква в обхождането трябва да пресметнем с каква стойност ще умножим позицията ѝ, например чрез използването на **switch** конструкция.

```
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;
}
```

След като имаме стойността на дадената буква, следва да я **умножим по позицията ѝ**. Тъй като индексите в стринга се различават с 1 от реалните позиции,

т.е. индекс 0 е позиция 1, индекс 1 е позиция 2 и т.н., ще добавим 1 към индексите.

```
weight += multiplier * (i + 1);
```

Всички получени междинни резултати трябва да бъдат добавени към обща сума за всяка една буква от 5-буквената комбинация.

Оформяне на изхода

Дали дадена дума трябва да се принтира, се определя по нейната тежест. Трябва ни условие, което да определи дали текущата тежест е в интервала [начало ... край], подаден ни на входа в началото на програмата. Ако това е така, принтираме **пълната дума** (**fullWord**).

Внимавайте да не принтирате думата от унि�кални букви. Тя ни бе необходима само за пресмятане на тежестта!

Думите са разделени с интервал и ще ги натрупваме в междинна променлива **result**, която е дефинирана като празен низ в началото.

```
if (weight >= firstNumber && weight <= secondNumber)
{
    result += fullWord + " ";
}
```

Финални щрихи

Условието е изпълнено с изключение случаите, в които нямаме нито една дума в подадения интервал. За да разберем дали сме намерили такава дума, можем просто да проверим дали низът **result** има началната си стойност (а именно празен низ), ако е така - отпечатваме **No**, иначе печатаме целия низ без последния интервал, използвайки метода **.Trim()**.

```
if (result == string.Empty)
{
    Console.WriteLine("No");
}
else
{
    Console.WriteLine(result.Trim());
}
```

Тестване в Judge системата

Тествайте решението си: <https://judge.softuni.bg/Contests/Practice/Index/518#2>.

Глава 9.2. Задачи за шампиони – част II

В тази глава ще разгледаме още няколко задачи, които причисляваме към категорията "за шампиони", т.е. по-трудни от стандартните задачи в тази книга.

По-сложни задачи върху изучавания материал

Преди да преминем към конкретните задачи, трябва да поясним, че те могат да се решат по-лесно с **допълнителни знания за програмирането и езика C#** (методи, масиви, колекции, рекурсия и т.н.), но всяко едно решение, което ще дадем сега, ще използва единствено материал, покрит в тази книга. Целта е да се научите да съставяте **по-сложни алгоритми** на базата на сегашните си знания.

Задача: дни за страстно пазаруване

Лина има истинска страст за пазаруване. Когато тя има малко пари, веднага отива в първия голям търговски център (мол) и се опитва да изхарчи възможно най-много за дрехи, чанти и обувки. Но любимото ѝ нещо са зимните намаления. Нашата задача е да анализираме странното ѝ поведение и да **изчислим покупките**, които Лина прави, когато влезе в мола, както и **парите, които ѝ остават**, когато приключи с пазаруването си.

На **първия ред** от входа ще бъде подадена **сумата**, която Лина има **преди** да започне да пазарува. След това при получаване команда "mall.Enter", Лина влиза в мола и започва да пазарува, докато не получи команда "mall.Exit". Когато Лина започне да пазарува, **на всяка линия** от входа ще получите стрингове, които представляват **действия**, които **Лина изпълнява**. Всеки **символ** в стринга представлява **покупка или друго действие**. Стинговите команди могат да съдържат само символи от **ASCII таблицата**. ASCII кода на всеки знак има **връзка с това колко Лина трябва да плати** за всяка стока. Интерпретирайте символите по следния начин:

- Ако символът е **главна буква**, Лина получава **50% намаление**, което означава, че трябва да намалите парите, които тя има, с 50% от цифровата репрезентация на символа от ASCII таблицата.
- Ако символът е **малка буква**, Лина получава **70% намаление**, което означава, че трябва да намалите парите, които тя има, с 30% от цифровата репрезентация на символа от ASCII таблицата.
- Ако символът е **"%"**, Лина прави **покупка**, която намалява парите ѝ на половина.
- Ако символът е **"***", Лина **изтегля пари от дебитната си карта** и добавя към наличните си средства 10 лева.
- Ако символът е **различен от упоменатите горе**, Лина просто прави покупка без намаления и в такъв случай просто извадете стойността на символа от ASCII таблицата от наличните ѝ средства.

Ако някоя от стойностите на покупките е по-голяма от текущите налични средства, Лина НЕ прави покупката. Парите на Лина не могат да бъдат по-малко от 0.

Пазаруването завършва, когато се получи команда **"`mall.Exit`"**. Когато това стане, трябва да принтирате броя на извършени покупки и парите, които са останали на Лина.

Входни данни

Входните данни трябва да се четат от конзолата. На първия ред от входа ще бъде подадена **сумата**, която Лина има преди да започне да пазарува. На всеки следващ ред ще има определена команда. Когато получите команда **"`mall.Enter`"**, на всеки следващ ред ще получавате **стрингове, съдържащи информация относно покупките/действията**, които Лина иска да направи. Тези стрингове ще продължат да бъдат подавани, докато не се получи команда **"`mall.Exit`"**. Винаги ще се подава само една команда **"`mall.Enter`"** и само една команда **"`mall.Exit`"**.

Изходни данни

Изходните данни трябва да се **принтират на конзолата**. Когато пазаруването приключи, на конзолата трябва да се принтира определен изход в зависимост от това какви покупки са били направени.

- Ако не са били направени никакви покупки – "No purchases. Money left: {останали пари} lv."
- Ако е направена поне една покупка – "{брой покупки} purchases. Money left: {останали пари} lv."

Парите трябва да се принтират с **точност от 2 символа** след десетичния знак.

Ограничения

- Парите са число с **плаваща запетая** в интервала: $[0 - 7.9 \times 10^{28}]$.
- Броят стрингове между **"`mall.Enter`"** и **"`mall.Exit`"** ще в интервала: **[1-20]**.
- Броят на символи във всеки стринг, който представлява команда, ще е в интервала: **[1-20]**.
- Позволено време за изпълнение: **0.1 секунди**.
- Позволена памет: **16 MB**.

Примерен вход и изход

| Вход | Изход | Коментар |
|---|--|--|
| 110 <code>mall.Enter</code> d <code>mall.Exit</code> | 1 purchases. Money left: 80.00 lv. | 'd' има ASCII код 100. 'd' е малка буква и за това Лина получава 70% отстъпка и така тя харчи $30\% * 100 = 30$ лв. След покупката ѝ остават $110 - 30 = 80$ лв. |

| Вход | Изход | Вход | Изход |
|-------------------------------------|--|--|---------------------------------------|
| 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. |
| | | | |

Насоки и подсказки

Ще разделим решението на задачата на три основни части:

- Обработка на входа.
- Алгоритъм на решаване.
- Форматиране на изхода.

Нека разгледаме всяка една част в детайли.

Обработване на входа

Входът за нашата задача се състои от няколко компонента:

- На първия ред имаме всички пари, с които Лина ще разполага за пазаруването.
- На всеки следващ ред ще имаме някакъв вид команда.

Първата част от прочитането е тривиална:

```
decimal shoppingMoney = decimal.Parse(Console.ReadLine());
```

Но във втората има детайл, с който трябва да се съобразим. Условието гласи: следното:

Всеки следващ ред ще има определена команда. Когато получите командата **“mall.Enter”**, на всеки следващ ред ще получите стрингове, съдържащи информация относно покупките/действията, които Лина иска да направи.

Тук е моментът, в който трябва да се съобразим, че от **втория ред нататък** трябва да започнем да четем команди, но **едва след като получим** командата **“mall.Enter”**, трябва да започнем да ги обработваме. Как можем да направим това? Използването на **while** или **do-while** цикъл е добър избор. Ето примерно решение как можем да пропуснем всички команди преди получаване на командата **“mall.Enter”**:

```
string command = Console.ReadLine();
while (command != "mall.Enter")
{
    command = Console.ReadLine();
}
```

```
command = Console.ReadLine();
```

Тук е мястото да отбележим, че извикването на `Console.ReadLine()` след края на цикъла се използва за преминаване към първата команда за обработване.

Алгоритъм за решаване на задачата

Алгоритъмът за решаването на самата задача е праволинеен – продължаваме да четем команди от конзолата, докато не бъде подадена командата “`mall.Exit`”. През това време обработваме всеки един знак (`char`) от всяка една команда спрямо правилата, указанi в условието, и едновременно с това модифицираме парите, които Лина има, и съхраняваме броя на покупките.

Нека разгледаме първите два проблема пред нашия алгоритъм. Първият проблем засяга начина, по който можем да четем командите, докато не срещнем “`mall.Exit`”. Решението, както видяхме по-горе, е да се използва **while-цикъл**. Вторият проблем е задачата да достъпим всеки един знак от подадената команда. Имайки предвид, че входните данни с командите представляват са от тип `string`, то най-лесният начин да достъпим всеки знак в тях е чрез **foreach цикъл**.

Ето как би изглеждало използване на два такива цикъла:

```
while (command != "mall.Enter")
{
    foreach (char action in command)
    {

    }

    command = Console.ReadLine();
}
```

Следващата част от алгоритъма ни е да обработим символите от командите, спрямо следните правила от условието:

- Ако символът е **главна буква**, Лина получава 50% намаление, което означава, че трябва да намалите парите, които тя има, с 50% от цифровата репрезентация ASCII символа.
- Ако символът е **малка буква**, Лина получава 70% намаление, което означава, че трябва да намалите парите, които тя има, с 30% от цифровата репрезентация ASCII символа.
- Ако символът е “%”, Лина прави покупка, която намалява парите ѝ на половина.
- Ако символът е “**”, Лина изтегля пари от дебитната си карта и добавя към наличните си средства 10 лева.

- Ако символът е **различен от упоменатите горе**, Лина просто прави покупка без намаления и в такъв случай просто извадете стойността на ASCII символа от наличните ѝ средства.

Нека разгледаме проблемите от първото условие, които стоят пред нас. Единият е как можем да разберем дали даден **символ представлява главна буква**. Можем да използваме един от двата начина:

- Имайки предвид, факта, че буквите в азбуката имат ред, можем да използваме следната проверка **`action >= 'A' && action <= 'Z'`**, за да проверим дали нашият символ се намира в интервала от големи букви.
- Можем да използваме функцията **`char.ToUpper(...)`**.

Другият проблем е как можем **да пропуснем даден символ**, ако той представлява операция, която изисква повече пари, отколкото Лина има? Това е възможно да бъде направено с използване на **`continue`** конструкцията.

Примерната проверка за първата част от условието изглежда по следния начин:

```
if (action >= 'A' && action <= 'Z')
{
    decimal price = action * 0.5m;
    if (shoppingMoney < price)
    {
        continue;
    }

    shoppingMoney -= price;
    purchases++;
}
```

Забележка: **`purchases`** е променлива от тип **`int`**, в която държим броя на всички покупки.

Смятаме, че читателят не би трябвало да изпита проблем при имплементацията на всички други проверки, защото са много сходни с първата.

Форматиране на изхода

Накрая ще **принтираме** определен **изход**, в зависимост от следното условие:

- Ако не са били направени никакви покупки – "**No purchases. Money left: {останали пари} lv.**"
- Ако е направена поне една покупка – "**{брой покупки} purchases. Money left: {останали пари} lv.**"

Операциите по принтиране са тривиални, като единственото нещо, с което трябва да се съобразим е, че **парите трябва да се принтират с точност от 2 символа** след десетичния знак.

Как можем да направим това? Ще оставим отговора на този въпрос на читателя.

Тестване в Judge системата

Тествайте решението си: <https://judge.softuni.bg/Contests/Practice/Index/519#0>.

Задача: числен израз

Бони е изключително могъща вещица. Тъй като силата на природата не е достатъчна, за да се бори успешно с вампири и върколаци, тя започнала да усвоява **силата на Изразите**. Изразът е много труден за усвояване, тъй като заклинанието разчита на способността за **бързо решаване на математически изрази**.

За използване на “Израз заклинание”, вещицата трябва да знае резултата от математически израз предварително. **Израз заклинанието** се състои от няколко прости математически израза. Всеки математически израз може да съдържа оператори за **събиране, изваждане, умножение и/или деление**.

Изразът се решава без да се вземат под внимание математическите правила при пресмятане на числови изрази. Това означава, че приоритет има последователността на операторите, а не това какъв вид изчисление правят. Изразът **може да съдържа скоби**, като **всичко в скобите се пресмята първо**. Всеки израз може да съдържа множество скоби, но не може да съдържа вложени скоби:

- Израз съдържащ (...(...)...) е невалиден.
- Израз съдържащ (...)...(...) е валиден.

Пример

Изразът

4 + 6 / 5 + (4 * 9 - 8) / 7 * 2

бива решен по следния начин:

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

Бони е много красива, но не чак толкова съобразителна, затова тя има нужда от нашата помощ, за да усвои силата на Изразите.

Входни данни

Входните данни се състоят от един ред, който бива подаван от конзолата – **математическият израз за пресмятане**. Редът **винаги завършва със символа "="**. Символът " $=$ " означава **край на математическия израз**.

Входните данни винаги са валидни и във формата, който е описан. Няма нужда да бъдат валидирани.

Изходни данни

Изходните данни трябва да се принтират на конзолата. Изходът се състои от един ред – **резултатът от пресметнатия математически израз**.

Резултатът трябва да бъде **закръглен до втората цифра след десетичния знак**.

Ограничения

- Изразите ще състоят от **максимум 2500 символа**.
- Числата от всеки математически израз ще са в интервала **[1...9]**.
- Операторите в математическите изрази винаги ще бъдат измежду **+** (събиране), **-** (изваждане), **/** (деление) или ***** (умножение).
- Резултатът от математическия израз ще е в интервала **[-100000.00...100000.00]**.
- Позволено време за изпълнение: **0.1 секунди**.
- Позволена памет: **16 MB**.

Примерен вход и изход

| Вход | Изход |
|----------------------|-------|
| $4+6/5+(4*9-8)/7*2=$ | 8.57 |

| Вход | Изход |
|----------------------------------|--------|
| $3+(6/5)+(2*3/7)*7/2*(9/4+4*1)=$ | 110.63 |

Насоки и подсказки

Както обикновено, първо ще прочетем и обработим входа, след това ще решим задачата и накрая ще отпечатаме резултата, форматиран, както се изисква.

Обработване на входа

Входните данни се състоят от точно един ред от конзолата. Тук имаме **два начина**, по които можем да обработим входа. Първият е чрез **прочитането на целия ред с командата Console.ReadLine()** и достъпването на всеки един символ (**char**) от реда чрез **foreach** цикъл. Вторият е чрез **прочитане на входа символ по символ** чрез **командата Console.Read()** и обработване на всеки символ.

За решаване на задачата ще използваме втория вариант.

```
int symbol = Console.Read();
```

Алгоритъм за решаване на задачата

За целите на нашата задача ще имаме нужда от две променливи:

- Една променлива, в която ще пазим **текущия резултат**.
- Още една променлива, в която ще пазим **текущия оператор** от нашия израз.

```
decimal result = 0;
int expressionOperator = '+';
```

Относно кода по-горе трябва да поясним два детайла. Първият е използването на тип **decimal** за съхранение на резултата на нашето уравнение с цел избягване на всякакви проблем с точността, които съществуват типовете **float** и **double**. Вторият е стойността по подразбиране на оператора – тя е **+**, за да може още първото срещнато число да бъде събрано с резултата ни.

След като вече имаме началните си променливи, трябва да помислим върху това каква ще е основната структура на нашата програма. От условието разбираме, че **всеки израз завършва с =**, т.е. ще трябва да четем и обработваме символи, докато не срещнем **=**. Следва точното изписване на **while** цикъл.

```
while (symbol != '=')
{
    symbol = Console.Read();
}
```

Следващата стъпка е обработването на нашата **symbol** променлива. За нея имаме три възможни случая:

- Ако символът е **начало на подизраз, заграден в скоби**, т.е. срещнатият символ е **(**.
- Ако символът е **цифра между 0 и 9**. Но как можем да проверим това? Как можем да проверим дали символът ни е цифра? Тук идва на помощ **ASCII кодът** на символа, чрез който можем да използваме следната формула: **[ASCII код на нашия символ] - [ASCII код на символа 0] = [цифрата, която репрезентира символа]**. Ако резултатът от тази проверка е между 0 и 9, то тогава нашият символ наистина е **ЧИСЛО**.
- Ако символът е **оператор**, т.е. е **+, -, *** или **/**.

```
if (symbol == '(')
{
}
```

```

else if (0 <= symbol - '0' && symbol - '0' <= 9)
{
}
else if (symbol == '+' ||
          symbol == '-' ||
          symbol == '/' ||
          symbol == '*')
{
}

```

Нека разгледаме действията, които трябва да извършим при съответните случаи, които дефинирахме:

- Ако нашият символ е **оператор**, единственото, което трябва да направим, е да зададем нова стойност на променливата **expressionOperator**.
- Ако нашият символ е **цифра**, тогава трябва да променим текущия резултат от израза в зависимост от текущия оператор, т.е. ако **expressionOperator** е `-`, тогава трябва да намалим резултата с цифровата репрезентация на текущия символ. Можем да вземем цифровата репрезентация на текущия символ, чрез формулата, която използвахме при проверката на този случай (**[ASCII код на нашия символ] - [ASCII код на символа 0] = [цифрата, която репрезентира символа]**).

Ето примерна имплементация на описаната идея:

```

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;
}

```

- Ако нашият символ е `(`, това индицира началото на подизраз (израз в скоби). По дефиниция подизразът трябва да се калкулира преди да се модифицира резултата от целия израз (действията в скобите се извършват първи). Това означава, че ще имаме локален резултат за подизраза ни и локален оператор.

```

if (symbol == '(')
{
    decimal innerResult = 0;
    int innerOperator = '+';
    symbol = Console.Read();
}

```

След това, за пресмятане стойността на подизраза използваме същите методи, които използвахме за пресмятане на главния израз – използваме **while цикъл**, за да четем символи (докато не срещнем символа `)`). В зависимост от това дали прочетения символ е цифра или оператор, модифицираме резултата на подизраза. Имплементацията на тези операции е аналогична на имплементацията за пресмятане на изрази, описана по-горе, затова смятаме, че читателят не би трябвало да има проблем с нея.

След като приключим калкулацията на резултата от подизраза ни, **модифицираме резултата на целия израз** в зависимост от стойността на `expressionOperator`.

```

switch (expressionOperator)
{
    case '+':
        result += innerResult;
        break;
    case '-':
        result -= innerResult;
        break;
    case '*':
        result *= innerResult;
        break;
    case '/':

```

```

    result /= innerResult;
    break;
}

```

Форматиране на изхода

Единствения изход, който програмата трябва да принтира на конзолата, е **резултатът от решаването на израза, с точност два символа след десетичния знак**. Как можем да форматираме изхода по този начин? Отговора на този въпрос оставяме на читателя.

Тестване в Judge системата

Тествайте решението си: <https://judge.softuni.bg/Contests/Practice/Index/519#1>.

Задача: Бикове и крави

Всички знаем играта „**Бикове и крави**“ (http://en.wikipedia.org/wiki/Bulls_and_cows). При дадено 4-цифreno **тайно** число и 4-цифreno **предполагаемо** число, използваме следните правила:

- Ако имаме цифра от предполагаемото число, която съвпада с цифра от тайното число и е на **същата позиция**, имаме **бик**.
- Ако имаме цифра от предполагаемото число, която съвпада с цифра от тайното число, но е **на различна позиция**, имаме **крава**.

| Тайно число | 1 | 4 | 8 | 1 | Коментар |
|---------------------|---|---|---|---|-------------------------|
| Предполагаемо число | 8 | 8 | 1 | 1 | Бикове = 1 Крави = 2 |
| Тайно число | 2 | 2 | 4 | 1 | Коментар |
| Предполагаемо число | 9 | 9 | 2 | 4 | Бикове = 0 Крави = 2 |

При дадено тайно число и брой на бикове и крави, нашата задача е **да намерим всички възможни предполагаеми числа** в нарастващ ред. Ако **не съществуват предполагаеми числа**, които да отговарят на зададените критерии на конзолата, трябва да се отпечата **"No"**.

Входни данни

Входните данни се четат от конзолата.

Входът се състои от 3 реда:

- Първият ред съдържа секретното число.
- Вторият ред съдържа броя бикове.
- Третият ред съдържа броя крави.

Входните данни ще бъдат винаги валидни. Няма нужда да бъдат проверявани.

Изходни данни

Изходните данни трябва да се принтират на конзолата.

Изходът трябва да се състои от **един единствен ред** – всички предполагаеми **числа**, разделени с единично празно място. Ако **не съществуват предполагаеми числа**, които да отговарят на зададените критерии на конзолата, трябва **да се изпише "No"**.

Ограничения

- Тайното число винаги ще се състои от **4 цифри в интервала [1..9]**.
- Броят на **кравите и биковете** винаги ще е в интервала **[0..9]**.
- Позволено време за изпълнение: **0.15 секунди**.
- Позволена памет: **16 MB**.

Примерен вход и изход

| Вход | Изход |
|----------------|--|
| 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 |

| Вход | Изход |
|----------------|---|
| 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 |

Насоки и подсказки

Ще решим задачата на няколко стъпки:

- Ще прочетем **входните данни**.
- Ще генерираме всички възможни **четирицифрени комбинации** (кандидати за проверка).

- За всяка генерирана комбинация ще изчислим **колко бика и колко крави** има в нея спрямо секретното число. При съвпадение с търсените бикове и крави, ще отпечатаме комбинацията.

Обработване на входа

За входа на нашата задача имаме 3 реда:

- Секретното число.
- Броят желани бикове.
- Броят желани крави.

Прочитането на тези входни данни е тривиално:

```
int guessNumber = int.Parse(Console.ReadLine());
int targetBulls = int.Parse(Console.ReadLine());
int targetCows = int.Parse(Console.ReadLine());
```

Алгоритъм за решаване на задачата

Преди да започнем писането на алгоритъма за решаване на нашия проблем, трябва да **декларираме флаг**, който да указва дали е намерено решение:

```
bool solutionFound = false;
```

Ако след приключването на нашия алгоритъм, този флаг все още е **false**, тогава ще принтираме **No** на конзолата, както е указано в условието.

```
if (!solutionFound)
{
    Console.WriteLine("No");
}
```

Нека започнем да размишляваме над нашия проблем. Това, което трябва да направим, е да **анализираме всички числа от 1111 до 9999** без тези, които съдържат в себе си нули (напр. **9011**, **3401** и т.н. са невалидни). Какъв е най-лесният начин за **генериране** на всички тези **числа**? С **вложени цикли**. Тъй като имаме **4-цифрен** число, ще имаме **4 вложени цикъла**, като всеки един от тях ще генерира **отделна** цифра от нашето число за тестване.

```
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++)
            {
```

Благодарение на тези цикли, имаме достъп до всяка една цифра на всички числа, които трябва да проверим. Следващата ни стъпка е да разделим секретното число на цифри. Това може да се постигне много лесно чрез комбинация от целочислено и модулно деление.

```
int guessDigit1 = (guessNumber / 1000) % 10;
int guessDigit2 = (guessNumber / 100) % 10;
int guessDigit3 = (guessNumber / 10) % 10;
int guessDigit4 = (guessNumber / 1) % 10;
```

Остават ни последните две стъпки преди да започнем да анализираме колко крави и бикове има в дадено число. Съответно, първата е **декларацията на counter променливи** във вложените ни цикли, за да броим кравите и биковете за текущото число. Втората стъпка е да направим **копия на цифрите на текущото число**, което ще анализираме, за да предотвратим проблеми с работата на вложите цикли, ако правим промени по тях.

```
int digitToCheck1 = digit1;
int digitToCheck2 = digit2;
int digitToCheck3 = digit3;
```

```
int digitToCheck4 = digit4;
```

```
int currentBulls = 0;
int currentCows = 0;
```

Вече сме готови да започнем анализирането на генерираните числа. Каква логика можем да използваме? Най-елементарният начин да проверим колко крави и бикове има в едно число е чрез **поредица от if-else проверки**. Да, не е най-оптималния начин, но с цел да не използваме знания извън пределите на тази книга, ще изберем този подход.

От какви проверки имаме нужда?

Проверката за бикове е елементарна – проверяваме дали **първата цифра** от генерираното число е еднаква със **същата цифра** от секретното число. Премахваме проверените цифри с цел да избегнем повторения на бикове и крави.

```
//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;
}

```

Повтаряме действието за втората, третата и четвъртата цифра.

Проверката за крави можем да направи по следния начин – първо проверяваме дали **първата цифра** от генерираното число **съвпада с втората, третата или четвъртата цифра** на секретното число. Примерна имплементация:

```

//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;
}

```

След това последователно проверяваме дали **втората цифра** от генерираното число **съвпада с първата, третата или четвъртата цифра** на секретното число, дали **третата цифра** от генерираното число съвпада с **първата, втората или четвъртата цифра** на секретното число и накрая проверяваме дали **четвъртата цифра** от генерираното число съвпада с **първата, втората или третата цифра** на секретното число.

Отпечатване на изход

След като приключим всички проверки, ни остава единствено да **проверим дали биковете и кравите в текущото генерирано число съвпадат с желаните бикове и крави, прочетени от конзолата**. Ако това е така, принтираме текущото число на конзолата.

```
if (currentBulls == targetBulls && currentCows == targetCows)
{
    if (solutionFound)
    {
        Console.Write(" ");
    }

    Console.WriteLine($"{digit1}{digit2}{digit3}{digit4}");
    solutionFound = true;
}
```

Тестване в Judge системата

Тествайте решението си: <https://judge.softuni.bg/Contests/Practice/Index/519#2>.

Глава 10. Методи

В настоящата глава ще се запознаем с **методи** и ще научим какво представляват те, както и кои са **базовите концепции** при работа с тях. Ще научим защо е **добра практика** да ги използваме, как да ги **декларираме и извикваме**. Ще се запознаем с **параметри и връщана стойност на метод**, както и как да използваме тази връщаща стойност. Накрая на главата, ще разгледаме **утвърдените практики** при използване на методите.

Какво е "метод"?

До момента установихме, че при **писане** на код на програма, която решава дадена задача, ни **улесява** това, че **разделяме** задачата на **части**. Всяка част отговаря за **дадено действие** и по този начин не само ни е **по-лесно** да решим задачата, но и значително се подобрява както **четимостта** на кода, така и проследяването за грешки.

Всяко едно парче код, което изпълнява дадена функционалност и което сме отделили логически, може да изземе функционалността на метода. Точно това представляват **методите – парчета код, които са именувани** от нас по определен начин и които могат да бъдат **извикани** толкова пъти, колкото имаме нужда.

Един метод може да бъде извикан толкова пъти, колкото ние преценим, че ни е нужно за решаване на даден проблем. Това ни **спестява** повторението на един и същи код няколко пъти, както и **намалява** възможността да пропуснем грешка при евентуална корекция на въпросния код.

Прости методи

Простите методи отговарят за изпълнението на дадено **действие**, което **спомага** за решаване на определен проблем. Такива действия могат да бъдат разпечатване на даден низ на конзолата, извършване на някаква проверка, изпълнение на цикъл и други.

Нека разгледаме следния **пример за прост метод**:

```
static void PrintHeader()
{
    Console.WriteLine("-----");
}
```

Този метод има задачата да отпечата заглавие, което представлява поредица от символа `-`. Поради тази причина името му е **PrintHeader**. Кръглите скоби (`и`) **винаги** следват името, независимо как сме именували метода. По-късно ще разгледаме как трябва да именуваме методите, с които работим, а за момента ще отбележим само, че е важно **името му да описва действието**, което той извършва.

Тялото на метода съдържа **програмния код**, който се намира между къдрявите скоби `{ и }`. Тези скоби **винаги** следват **декларацията** му и между тях поставяме кода, който решава проблема, описан от името на метода.

Защо да използваме методи?

До тук установихме, че методите спомагат за **разделянето на обемна задача на по-малки части**, което води до **по-лесно решаване** на въпросното задание. Това прави програмата ни не само по-добре структурирана и лесно четима, но и по-разбираема.

Чрез методите **избягваме повторението** на програмен код. **Повтарящият** се код е **лоша практика**, тъй като силно **затруднява поддръжката** на програмата и води до грешки. Ако дадена част от кода ни присъства в програмата няколко пъти и се наложи да променим нещо, то промените трябва да бъдат направени във всяко едно повторение на въпросния код. Вероятността да пропуснем място, на което трябва да нанесем корекция, е много голяма, което би довело до некоректно поведение на програмата. Това е причината, поради която е **добра практика**, ако използваме даден фрагмент код **повече от веднъж** в програмата си, да го **дефинираме като отделен метод**.

Методите ни предоставят **възможността** да използваме даден **код няколко пъти**. С решаването на все повече и повече задачи ще установите, че използването на вече съществуващи методи спестява много време и усилия.

Декларирани на методи

В езика C# **декларираме** методите в рамките на даден клас, т.е. между отварящата { и затваряща } скоби на класа. Декларирането представлява регистрирането на метода в програмата, за да бъде разпознаван в останалата част от нея. Найдобре познатият ни пример за метод е метода **Main(...)**, който използваме във всяка една програма, която пишем.

```
class Methods
{
    0 references
    static void Main()
    {
    }
}
```

Да разгледаме задължителните елементи в декларацията на един метод:

```
static double GetSquare(double num)
{
    return num * num;
}
```

- **Тип на връщаната стойност.** В случая типа е **double**, което означава, че методът от примера ще **върне резултат**, който е от тип **double**. Връщаната стойност може да бъде както **int**, **double**, **string** и т.н., така и **void**. Ако типът

е **void**, то това означава, че методът **не връща** резултат, а само **изпълнява** дадена операция.

- **Име на метода.** Името на метода е **определеното от нас**, като не забравяме, че трябва да **описва функцията**, която е изпълнявана от кода в тялото му. В примера името е **GetSquare**, което ни указва, че задачата на този метод е да изчисли лицето на квадрат.
- **Списък с параметри.** Декларира се между скобите **(и)**, които изписваме след името му. Тук изброяваме поредицата от **параметри**, които метода ще използва. Може да присъства **само един** параметър, **няколко** такива или да е **празен** списък. Ако няма параметри, то ще запишем само скобите **()**. В конкретния пример декларираме параметъра **double num**.
- Декларация **static** в описанието на метода. За момента може да приемем, че **static** се пише винаги, когато се декларира метод, а по-късно, когато се запознаем с обектно-ориентираното програмиране (ООП), ще разберем разликата между **статични методи** (споделени за целия клас) и **методи на обект**, които работят върху данните на конкретна инстанция на класа (обект).

При деклариране на методи е важно да спазваме **последователността** на основните му елементи – първо **тип на връщаната стойност**, след това **име на метода** и накрая **списък от параметри**, ограден с кръгли скоби **()**.

След като сме декларирали метода, следва неговата **имплементация** (**тяло**). В тялото на метода описваме **алгоритъма**, по който той решава даден проблем, т.е. тялото съдържа кода (програмен блок), който реализира **логиката** на метода. В показания пример изчисляваме лицето на квадрат, а именно **num * num** и след това връщаме като резултат изчислената стойност.

Когато декларираме дадена променлива в тялото на един метод, я наричаме **локална** променлива за метода. Областта, в която съществува и може да бъде използвана тази променлива, започва от реда, на който сме я декларирали и стига до затварящата къдрава скоба **}** на тялото на метода. Тази област се нарича **област на видимост** на променливата (variable scope).

Извикване на методи

Извикването на метод представлява **стартрирането на изпълнението на кода**, който се намира в **тялото на метода**. Това става като изпишем **името** му, последвано от кръглите скоби **()** и знака **;** за край на реда. Ако методът ни изисква входни данни, то те се подават в скобите **()**, като последователността на фактическите параметри трябва да съвпада с последователността на подадените при декларирането на метода. Ето един пример за **деклариране и извикване** на метод:

```
// declaring method
static void PrintHeader()
{
    Console.WriteLine("-----");
}
```

```
static void Main()
{
    // invoking the declared method
    PrintHeader();
}
```

Даден метод може да бъде извикан от **няколко места** в нашата програма. Единият начин е да бъде извикан от **главния метод**.

```
static void Main()
{
    // invoking the declared method
    // from the Main() method body
    PrintHeader();
}
```

Метод може да бъде извикан и от **тялото на друг метод**, който **не** е главния метод на програмата ни.

```
static void PrintHeader()
{
    // invoking methods from another method
    PrintHeaderTop();
    PrintHeaderBottom();
}
```

Съществува вариант методът да бъде извикан от **собственото си тяло**. Това се нарича **рекурсия** и можете да намерите повече информация за нея в Wikipedia (<https://bg.wikipedia.org/wiki/Рекурсия>) или да потърсите сами в Интернет.

Важно е да знаем, че ако един метод е деклариран в даден клас, то той може да бъде извикван преди реда, на който е деклариран.

Пример: празна касова бележка

Да се напише метод, който печата празна касова бележка. Методът трябва да извика други три метода: един за принтиране на заглавието, един за основната част на бележката и един за долната част.

| Част от касовата бележка | Текст |
|--------------------------|---------------------------------------|
| Горна част | CASH RECEIPT ----- |
| Средна част | Charged to _____ Received by _____ |

| Част от касовата бележка | Текст |
|--------------------------|----------------------|
| Долна част | ----- (c) SoftUni |

Примерен вход и изход

| Вход | Изход |
|--------|--|
| (няма) | CASH RECEIPT ----- Charged to_____ Received by_____ ----- (c) SoftUni |

Насоки и подсказки

Първата ни стъпка е да създадем **void** метод за **принтиране на заглавната част** от касовата бележка (header). Нека му дадем смислено име, което описва кратко и ясно задачата му, например **PrintReceiptHeader**. В тялото му ще напишем кода от примера по-долу:

```
static void PrintReceiptHeader()
{
    Console.WriteLine("CASH RECEIPT");
    Console.WriteLine("-----");
}
```

Съвсем аналогично ще създадем още два метода за **разпечатване на средната част** на бележката (тяло) **PrintReceiptBody** и за разпечатване на **долната част** на бележката (footer) **PrintReceiptFooter**.

След това ще създадем и **още един метод**, който ще извиква трите метода, които написахме до момента един след друг:

```
static void PrintReceipt()
{
    PrintReceiptHeader();
    PrintReceiptBody();
    PrintReceiptFooter();
}
```

Накрая ще **извикаме** метода **PrintReceipt** от тялото на главния **Main** метод за нашата програма:

```
static void Main()
{
    PrintReceipt();
}
```

Тестване в Judge системата

Програмата с общо пет метода, които се извикват един от друг, е готова и можем да я изпълним и тестваме, след което да я пратим за проверка в judge системата: <https://judge.softuni.bg/Contests/Practice/Index/594#0>.

Методи с параметри

Много често в практиката, за да бъде решен даден проблем, методът, с чиято помощ постигаме това, се нуждае от **допълнителна информация**, която зависи от задачата му. Именно тази информация представляват **параметрите на метода** и неговото поведение зависи от тях.

Използване на параметри в методите

Както отбелязахме по-горе, параметрите освен нула на брой, могат също така да са **един или няколко**. При декларацията им ги разделяме със запетая. Те могат да бъдат от всеки един тип (**int, string** и т.н.), а по-долу е показан пример как точно ще бъдат използвани от метода.

Декларираме метода и списъка му с **параметри**, след което пишем кода, който той ще изпълнява.

```
static void PrintNumbers(int start, int end)
{
    for (int i = start; i <= end; i++)
    {
        Console.WriteLine("{0} ", i);
    }
}
```

След това извикваме метода и му подаваме конкретни стойности:

```
static void Main()
{
    PrintNumbers(5, 10);
}
```

При **декларирането на параметри** можем да използваме **различни** типове променливи, като трябва да внимаваме всеки един параметър да има **тип и име**. Важно е да отбележим, че при последващото извикване на метода, трябва да подаваме **стойности** за параметрите по **реда**, в който са **декларирали** самите те. Ако

имаме подадени параметри в реда **int** и след това **string**, при извикването му не можем да подадем първо стойност за **string** и след това за **int**. Единствено можем да разменяме местата на подадените параметри, ако изрично изпишем преди това името на параметъра, както ще забележим малко по-нататък в един от примерите. Това като цяло не е добра практика!

Нека разгледаме примера за декларация на метод, който има няколко параметъра от различен тип.

```
static void PrintStudent(string name, int age, double grade)
{
    Console.WriteLine("Student: {0}; Age: {1}, Grade: {2}",
        name, age, grade);
}
```

Пример: знак на цяло число

Да се създаде метод, който печата знака на цяло число n.

Примерен вход и изход

| Вход | Изход |
|------|----------------------------|
| 2 | The number 2 is positive. |
| -5 | The number -5 is negative. |
| 0 | The number 0 is zero. |

Насоки и подсказки

Първата ни стъпка е **създаването** на метод и даването му на описателно име, например **PrintSign**. Този метод ще има само един параметър от тип **int**.

```
static void PrintSign(int n)
{
}
```

Следващата ни стъпка е **имплементирането** на логиката, по която програмата ще проверява какъв точно е знакът на числото. От примерите виждаме, че има **три случая** – числото е по-голямо от нула, равно на нула или по-малко от нула, което означава, че ще направим три проверки в тялото на метода. Оставяме имплементацията на читателя.

Следващата ни стъпка е да прочетем входното число и да извикаме новия метод от тялото на **Main()** метода.

```
static void Main()
{
    int n = int.Parse(Console.ReadLine());
    PrintSign(n);
}
```

Тестване в Judge системата

Тествайте решението си: <https://judge.softuni.bg/Contests/Practice/Index/594#1>.

Незадължителни параметри

Езикът C# поддържа използването на **незадължителни** параметри. Те позволяват пропускането на параметри при извикването на метода. Декларирането им става чрез **осигуряване на стойност по подразбиране** в описанието на съответния параметър.

Следващият пример онагледява употребата на незадължителните параметри:

```
static void PrintNumbers(int start = 0, int end = 100)
{
    for (int i = start; i <= end; i++)
        Console.Write("{0} ", i);
}
```

Показаният метод **PrintNumbers** може да бъде извикан по няколко начина, както се вижда от примера по-долу:

```
static void Main()
{
    PrintNumbers(5, 10);
    PrintNumbers(15);
    PrintNumbers();
    PrintNumbers(end: 40, start: 35);
}
```

Пример: принтиране на триъгълник

Да се създаде метод, който принтира триъгълник, както е показано в примерите.

Примерен вход и изход

| Вход | Изход | Вход | Изход |
|------|-------------------------------|------|---|
| 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 |

Насоки и подсказки

Преди да създадем метод за принтиране на един ред с дадени начало и край, прочитаме входното число от конзолата. След това избираме съмислено име за метода, което описва целта му, например **PrintLine**, и го имплементираме.

```
static void PrintLine(int start, int end)
{
    for (int i = start; i <= end; i++)
    {
        Console.WriteLine(i + " ");
    }
    Console.WriteLine();
}
```

От задачите за рисуване на конзолата си спомняме, че е добра практика да разделяме фигурата на няколко части. За наше улеснение ще разделим триъгълника на три части – горна, средна линия и долната.

Следващата ни стъпка е с цикъл да разпечатаме **горната половина** от триъгълника:

```
for (int i = 0; i < n; i++)
{
    PrintLine(1, i);
}
```

След това разпечатваме **средната линия**:

```
PrintLine(1, n);
```

Накрая разпечатваме **долната част** от триъгълника, като този път стъпката на цикъла намалява:

```
for (int i = n - 1; i > 0; i--)
{
    PrintLine(1, i);
}
```

Тестване в Judge системата

Тествайте решението си: <https://judge.softuni.bg/Contests/Practice/Index/594#2>.

Пример: рисуване на запълнен квадрат

Да се нарисува на конзолата запълнен квадрат със страна n, както е показано в примерите.

Примерен вход и изход

| Вход | Изход | Вход | Изход | Вход | Изход |
|------|--|------|--|------|----------------|
| 4 | ----- - \ \ \ / - - \ \ \ / - ----- | 3 | ----- - \ \ / - - \ \ / - ----- | 2 | ----- ----- |
| | | | | | |

Насоки и подсказки

Първата ни стъпка е да прочетем входа от конзолата. След това трябва да създадем метод, който ще принтира първия и последен ред, тъй като те са еднакви. Нека не забравяме, че трябва да му дадем **описателно име** и да му зададем като параметър дължината на страната. Ще използваме конструктора **new string**.

```
static void PrintHeaderFooter(int n)
{
    Console.WriteLine(new string('-', 2 * n));
}
```

Следващата ни стъпка е да създадем метод, който ще рисува на конзолата средните редове. Отново задаваме описателно име, например **PrintMiddleRow**.

```
static void PrintMiddleRow(int n)
{
    Console.Write("-");
    for (int i = 0; i < n - 1; i++)
    {
        Console.Write("\\" "/");
    }
    Console.Write("-");
    Console.WriteLine();
}
```

На края извикваме създадените методи в главния метод **Main()** на програмата, за да нарисуваме целия квадрат:

```
static void Main()
{
    int input = int.Parse(Console.ReadLine());
    PrintHeaderFooter(input);
    // TODO: Draw the rest of the square
}
```

Тестване в Judge системата

Тествайте решението си: <https://judge.softuni.bg/Contests/Practice/Index/594#3>.

Връщане на резултат от метод

До момента разглеждахме методи, които извършват дадено действие, например отпечатване на даден текст, число или фигура на конзолата. Освен този тип методи, съществуват и такива, които могат да **връщат** някакъв **резултат**. Именно тези методи ще разгледаме в следващите редове.

Типове на връщаната от метода стойност

До сега разглеждахме примери, в които при декларация на методи използвахме ключовата дума **void**, която указва, че методът **не** връща резултат, а изпълнява определено действие.

```
static void AddOne(int n)
{
    n += 1;
    Console.WriteLine(n);
}
```

Ако **заместим void** с тип на променлива, то това ще укаже на програмата, че метода трябва да върне някаква стойност от указанния тип. Тази върната стойност може да бъде от всякачъв тип – **int, string, double** и т.н.



За да върне един метод **резултат** е нужно да внимаваме да напишем очаквания тип на резултата при декларацията на метода на мястото на **void**.

```
static int PlusOne(int n)
{
    return n + 1;
}
```

Важно е да отбележим, че **результатът**, който се връща от метода, може да е от тип, съвместим с типа на връщаната стойност на метода. Например, ако декларираният тип на връщаната стойност е **double**, то можем да върнем **int** резултат.

Оператор return

За да получим резултат от метода, на помощ идва операторът **return**. Той трябва да бъде използван в тялото на метода и указва на програмата да спре изпълнението му и да върне на извиквача на метода определена **стойност**, която се определя от израза след въпросния оператор **return**.

В примера по-долу имаме метод, който чете две имена от конзолата, съединява ги и ги връща като резултат. Връщаната стойност е от тип **string**:

```
static string ReadFullName()
{
    string firstName = Console.ReadLine();
    string lastName = Console.ReadLine();
    return firstName + " " + lastName;
}
```

Операторът **return** може да бъде използван и във **void** методи. Тогава самият метод ще спре изпълнението си, без да връща никаква стойност, а след него не трябва да има израз, който да бъде върнат. В този случай употребата на **return** е единствено за излизане от метода.

Има случаи, в които **return** може да бъде извикван от няколко места в метода, но само ако има **определенни** входни условия.

В примера по-долу имаме метод, който сравнява две числа и връща резултат съответно **-1**, **0** или **1** според това дали първият аргумент е по-малък, равен или по-голям от втория аргумент, подаден на функцията. Методът използва ключовата дума **return** на три различни места, за да върне три различни стойности според логиката на сравненията на числата:

```
static int CompareTo(int number1, int number2)
{
    if (number1 > number2)
    {
        return 1;
    }
    else if (number1 == number2)
    {
        return 0;
    }
    else
    {
```

```

        return -1;
    }
}

```

Кодът след `return` е недостъпен

След `return` оператора, в текущия блок, **не** трябва да има други редове код, тъй като тогава Visual Studio ще покаже предупреждение, съобщавайки ни, че е засягъл код, който **не може да бъде достъпен**:

```

static int SomeFunction()
{
    return 5;
    return 2; // This code is unreachable
}

```



The screenshot shows a code editor with the above code. A green squiggly underline is under the second `return 2;` line. A tooltip window titled "Unreachable code detected" appears below it, containing the text "Show potential fixes (Alt+Enter or Ctrl+.)".



В програмирането не може да има два пъти оператор `return` един след друг, защото изпълнението на първия няма да позволи да се изпълни вторият. Понякога програмистите се шегуват с фразата **"пиши `return; return;` и да си ходим"**, за да обяснят, че логиката на програмата е объркана.

Употреба на връщаната от метода стойност

След като даден метод е изпълнен и върне стойност, то тази стойност може да се използва по **няколко** начина.

Първият е да **присвоим резултата на променлива** от съвместим тип:

```
int max = GetMax(5, 10);
```

Вторият е **резултатът да бъде използван в израз**:

```
decimal total = GetPrice() * quantity * 1.20m;
```

Третият е да **подадем** резултата от работата на метода към **друг метод**:

```
int age = int.Parse(Console.ReadLine());
```

Пример: пресмятане на лицето на триъгълник

Да се напише метод, който изчислява лицето на триъгълник по дадени основа и височина и връща стойността му.

Примерен вход и изход

| Вход | Изход |
|------|-------|
| 3 | |
| 4 | 6 |

Насоки и подсказки

Първата ни стъпка е да прочетем входа. След това **създаваме** метод, но този път внимаваме при **декларацията** да подадем коректния **тип** данни, които искаме метода да върне, а именно **double**.

```
static double GetTriangleArea(double length,
    double height)
{
    return (length * height) / 2;
}
```

Следващата ни стъпка е да **извикаме** новия метод от нашия **Main()** метод и да запишем върнатата стойност в подходяща променлива и накрая да я отпечатаме.

Ето и примерна реализация:

```
static void Main()
{
    double a = double.Parse(Console.ReadLine());
    double h = double.Parse(Console.ReadLine());
    double area = GetTriangleArea(a, h);
    Console.WriteLine(area);
}
```

Тестване в Judge системата

Тествайте решението си: <https://judge.softuni.bg/Contests/Practice/Index/594#4>.

Пример: степен на число

Да се напише метод, който изчислява и връща резултата от повдигането на число на дадена степен.

Примерен вход и изход

| Вход | Изход |
|------|-------|
| 2 | |
| 8 | 256 |

| Вход | Изход |
|------|-------|
| 3 | |
| 4 | 81 |

Насоки и подсказки

Първата ни стъпка отново ще е да прочетем входните данни от конзолата. Следващата стъпка е да създадем метод, който ще приема два параметъра (числото и степента) и ще връща като резултат число от тип **double**.

```
static double CalculatePower(double number, double power)
{
    double result = 0d;
    // TODO: Calculate result
    // Use a loop or Math.Pow(...)
    return result;
}
```

След като сме направили нужните изчисления, ни остава да разпечатаме резултата в главния метод **Main()** на програмата.

Тестване в Judge системата

Тествайте решението си: <https://judge.softuni.bg/Contests/Practice/Index/594#5>.

Методи, връщащи няколко стойности

В практиката се срещат случаи, в които се нуждаем даден метод да върне повече от един елемент като резултат. За да е възможен подобен сценарий във Visual Studio и C# (от C# 7 нататък) е интегриран стойностният тип **ValueTuple**, както и литерал от тип **ValueTuple**. Накратко типът **ValueTuple** представлява съвкупност от две стойности, позволяващи временното съхранение на **множество стойности**. Стойностите биват съхранявани в променливи (полета – какво са полета, ще разгледаме на по-късен етап) от съответните типове. Въпреки, че типът **Tuple** съществува и преди C# 7, той не е добре поддържан от езика в предишните му версии и е неефективен. Затова в предходните версии на езика C# елементите в един **Tuple** са представяни като **Item1**, **Item2** и т.н. и имената на техните променливи (променливите, в които се съхраняват) е било невъзможно да бъдат променяни. В C# 7 е въведена поддръжка на типа (**ValueTuple**), което позволява задаване на смислови имена на елементите в един **ValueTuple**.

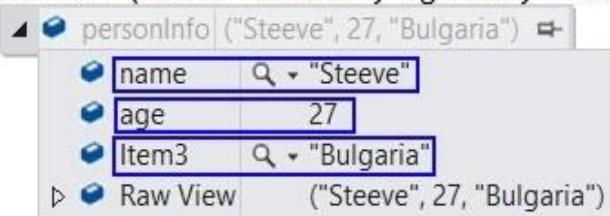
Деклариране на ValueTuple

Нека разгледаме примерна декларация на променлива от тип **ValueTuple**:

```
var personInfo = (name: "Steeve", age: 27, "Bulgaria");
```

За улеснение при декларирането използваме ключовата дума **var**, а в скобите изброяваме **имената на желаните стойности**, следвани от **самите стойности**. Нека погледнем и в дебъг режим какво се съдържа в променливата **personInfo**:

```
var personInfo = (name: "Steeve", age: 27, "Bulgaria");
```



Виждаме, че се състои от няколко **полета с имена и стойности**, описани при инициализацията на променливата. Забелязваме, че последната променлива е именувана **Item3**. Това е така, защото по време на инициализацията не сме уточнили име за променливата, в която се пази стойността **"Bulgaria"**. В такъв случай именуването е **по подразбиране**, т.е. променливите са именувани с **Item1**, **Item2**, **Item3** и т.н.

Метод, връщащ няколко стойности

Този метод приема за параметри две цели числа (**x** и **y**) и **връща две стойности** – резултата от целочислено деление на двете числа и остатъка от делението им:

```
static (int result, int remainder) Divide(int x, int y)
{
    int result = x / y;
    int remainder = x % y;

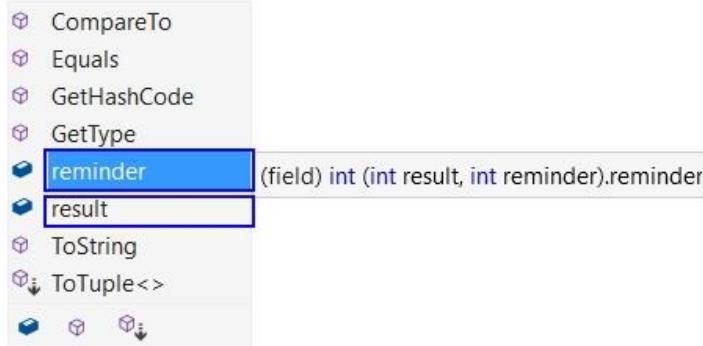
    return (result, remainder);
}
```

Този метод връща резултат от тип **ValueTuple**, съдържащ две променливи (полета) от тип **int**, съответно именувани **result** и **remainder**. Извикването на метода се осъществява по следния начин:

```
var division = Divide(1, 3);
```

За да достъпим резултатите, върнати от метода, прилагаме **точковата нотация** към променливата **division**:

```
division.
```



Варианти на методи

В много езици за програмиране един и същ метод може да е деклариран в **няколко варианта** с еднакво име и различни параметри. Това е известно с термина “method overloading”. Сега нека разгледаме как се пишат тези overloaded methods.

Сигнатура на метода

В програмирането **начинът**, по който се **идентифицира** един метод, е чрез **двойката елементи** от декларацията му – **име** на метода и **списък** от неговите параметри. Тези два елемента определят неговата **спецификация**, т. нар. **сигнатура** на метода.

В този пример сигнатурата на метода е неговото име (**Print**), както и неговият параметър (**string text**):

```
static void Print(string text)
{
    Console.WriteLine(text);
}
```

Ако в програмата ни има **методи с еднакви имена**, но с различни сигнатури, то казваме, че имаме **варианти на методи** (method overloading).

Варианти на методи

Както споменахме, ако използваме **едно и също име** за **няколко метода с различни сигнатури**, то това означава, че имаме **варианти на метод**. Кодът по-долу показва как три различни метода могат да са с едно и също име, но да имат различни сигнатури и да изпълняват различни действия.

```
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);
}
```

Сигнатура и тип на връщаната стойност

Важно е да отбележим, че **връщаният тип като резултат** на метода не е част от **сигнатурата му**. Ако връщаната стойност беше част от сигнатурата на метода, то няма как компилаторът да знае кой метод точно да извика.

Нека разгледаме следния пример – имаме два метода с различен тип на връщаната стойност. Въпреки това Visual Studio ни показва, че има грешка, защото сигнатурите и на двата са еднакви. Съответно при опит за извикване на метод с име **Print(...)**, компилаторът не би могъл да прецени кой от двата метода да изпълни.

```
static void Print(string text)
{
    Console.WriteLine(text);
}

static string Print(string text)
{
    return text;
}
```

Пример: по-голямата от две стойности

Като входни данни са дадени две стойности от един и същ тип. Стойностите могат да са от тип **int**, **char** или **string**. Да се създаде метод **GetMax()**, който връща като резултат по-голямата от двете стойности.

Примерен вход и изход

| Вход | Изход |
|------|-------|
| int | |
| 2 | 16 |
| 16 | |

| Вход | Изход |
|------|-------|
| char | |
| a | z |
| z | |

| Вход | Изход |
|--------|-------|
| string | |
| Ivan | Todor |
| Todor | |

Насоки и подсказки

За да създадем този метод, първо трябва да създадем три други метода с едно и също име и различни сигнтури. Първо създаваме метод, който ще сравнява цели числа.

```
static int GetMax(int first, int second)
{
    if (first >= second)
    {
        // TODO: return value
    }
    // TODO: handle other cases
}
```

Следвайки логиката от предходния метод, създаваме такъв със същото име, който обаче ще сравнява символи:

```
static char GetMax(char first, char second)
{
    // TODO: create logic
}
```

Следващият метод, който трябва да създадем, ще сравнява низове. Тук логиката ще е малко по-различна, тъй като стойностите от тип **string** не позволяват да бъдат сравнявани чрез операторите **<** и **>**.

Ще използваме метода **CompareTo(...)**, който връща чисрова стойност: по-голяма от 0 (сравняваният обект е по-голям), по-малка от 0 (сравняваният обект е по-малък) и 0 (при два еднакви обекта).

```
static string GetMax(string first, string second)
{
    if (first.CompareTo(second) >= 0)
    {
        // TODO: return value
    }
    // TODO: return value
}
```

Последната стъпка е да прочетем входните данни, да използваме подходящи променливи и да извикаме метода **GetMax()** от тялото на метода **Main()**:

```
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
    }
}

```

Тестване в Judge системата

Тествайте решението си: <https://judge.softuni.bg/Contests/Practice/Index/594#6>.

Вложени методи (локални функции)

Нека разгледаме следния пример:

```

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));
}

```

Какво е локална функция (локален метод)?

Виждаме, че в този код, в главния метод **Main()** има **друг** деклариран метод **Result()**. Такъв **вложен** метод се нарича **локална** функция и е нововъведение в C# 7. Локалните функции могат да се декларират във всеки един друг метод. Когато C# компилаторът компилира такива функции, те биват превърнати в **private** методи. Тъй като разликата между **public** и **private** методи се изучава на по-късен етап, за момента ще отбележим, че **private** методите могат да се използват само в класа, в който са декларирани. Програмите, които пишем на

това ниво, използват само един клас, затова и приемаме, че можем да използваме вложените методи без каквото и да било притеснения.

Защо да използваме локални функции?

С времето и практиката ще открием, че когато пишем код, често се нуждаем от методи, които бихме използвали само един път, или пък нужният ни метод става твърде дълъг. По-нагоре споменахме, че когато един метод съдържа в себе си прекалено много редове код, то той става труден за поддръжка и четене. В тези случаи на помощ идват локалните функции – те предоставят възможност в даден метод да се декларира друг метод, който ще бъде използван например само един път. Това спомага кода ни да е по-добре подреден и по-лесно четим, което от своя страна спомага за по-бърза корекция на евентуална грешка в кода и намалява възможността за грешки при промени в програмната логика.

Декларирале на локални методи (функции)

Нека отново разгледаме примера от по-горе.

```
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));
}
```

В този пример, методът **Result()** е локална функция, тъй като е вложен в метода **Main()**, т.е. **Result()** е локален за **Main()**. Това означава, че методът **Result()** може да бъде използван само в метода **Main()**, тъй като е деклариран в него. Единствената разлика между вложените методи и обикновените методи е, че вложените методи не могат да бъдат **static**. Тъй като дефиницията за **static** метод се разглежда на по-късен етап, за момента ще приемем, че при декларирането на една локална функция, изписваме единствено типа на връщаната стойност, името на метода и списъка му с параметри. В конкретния разглеждан случай, това е **double Result(double a, double b)**.

Локалните функции имат достъп до променливи, които се използват в съдържащия ги метод. Следващият пример демонстрира как се случва това. Тази особеност на вложените методи спестява време и код, които иначе бихме вложили, за да предаваме на вложените методи параметри.

```

static void Main()
{
    string output = "I am a local function";

    void PrintOutput()
    {
        Console.WriteLine(output);
    }
}

```

Именуване на методи и утвърдени практики

В тази част ще се запознаем с някои **утвърдени практики** при работа с методи, свързани с именуването, подредбата на кода и неговата структура.

Именуване на методи

Когато именуваме даден метод е препоръчително да използваме **смислени имена**. Тъй като всеки метод **отговаря** за някаква част от нашия проблем, то при именуването му трябва да вземем предвид **действието, което той извършва**, т.е. добра практика е **името да описва неговата цел**.

Задължително е името да започва с **главна буква** и трябва да е съставено от глагол или от двойка: глагол + съществително име. Форматирането на името става, спазвайки **Upper Case Camel** конвенцията, т.е. **всяка дума, включително първата, започва с главна буква**. Кръглите скоби (**и**) винаги следват името му.

Всеки метод трябва да изпълнява самостоятелна задача, а името на метода трябва да описва каква е неговата функция.

Няколко примера за **коректно** именуване на методи:

- **FindStudent**
- **LoadReport**
- **Sine**

Няколко примера за **лошо** именуване на методи:

- **Method1**
- **DoSomething**
- **HandleStuff**
- **SampleMethod**
- **DirtyHack**

Ако не можем да измислим подходящо име, то най-вероятно методът решава повече от една задача или няма ясно дефинирана цел и тогава трябва да помислим как да го разделим на няколко отделни метода.

Именуване на параметрите на методите

При именуването на **параметрите** на метода важат почти същите правила, както и при самите методи. Разликите тук са, че за имената на параметрите е добре да използваме съществително име или двойка от прилагателно и съществително име, както и че при именуване на параметрите се спазва **lowerCamelCase** конвенцията, т.е. **всички думи без първата започват с главна буква**. Трябва да отбележим, че е добра практика името на параметъра да **указва** каква е **мерната единица**, която се използва при работа с него.

Няколко примера за **коректно** именуване на параметри на методи:

- **firstName**
- **report**
- **speedKmH**
- **usersList**
- **fontSizeInPixels**
- **font**

Няколко примера за **некоректно** именуване на параметри:

- **p**
- **p1**
- **p2**
- **populate**
- **LastName**
- **last_name**

Добри практики при работа с методи

Нека отново припомним, че един метод трябва да изпълнява **само една** точно определена **задача**. Ако това не може да бъде постигнато, тогава трябва да помислим как да **разделим** метода на няколко отделни такива. Както казахме, името на метода трябва точно и ясно да описва неговата цел. Друга добра практика в програмирането е да **избягваме** методи, по-дълги от екрана ни (приблизително). Ако все пак кода стане много обемен, то е препоръчително метода да се **раздели** на няколко по-кратки, както в примера по-долу.

```
static void PrintReceipt()
{
    PrintHeader();
    PrintBody();
    PrintFooter();
}
```

Структура и форматиране на кода

При писането на методи трябва да внимаваме да спазваме коректна **индентация** (отместване по-навътре на блокове от кода).

Пример за **правилно** форматиран C# код:

```
static void Main()
{
    // some code here...
    // some more code...
}
```

Пример за **некоректно** форматиран C# код:

```
static void Main() {
    // some code...
// some more code...
}
```

Друга добра практика при писане на код е да **оставяме празен ред** между методите, след циклите и условните конструкции.

Също така, опитвайте да избягвате да пишете **дълги редове и сложни изрази**. С времето ще установите, че това подобрява четимостта на кода и спестява време.

Препоръчваме винаги да се **използват къдреви скоби за тялото на проверки и цикли**. Скобите не само подобряват четимостта, но и намалят възможността да бъде допусната грешка и програмата ни да се държи некоректно.

Какво научихме от тази глава?

В тази глава се запознахме с базовите концепции при работа с методи:

- Научихме, че **целта** на методите е да **разделят** големи програми с много редове код на по-малки и кратки задачи.
- Запознахме се със **структурата** на методите, как да ги **декларираме** и **извикваме** по тяхното име.
- Разгледахме примери за методи с **параметри** и как да ги използваме в нашата програма: как да подаваме стойности към метод.
- Научихме как да връщаме резултат от работата на метод, какво представляват **сигнатурата** и **връщаната стойност** на метода, както и каква е функцията на оператора **return** в методите.
- Запознахме се с **утвърдени практики** при работа с методи в програмирането: как да именуваме методите и техните параметри, как да структурираме и форматираме кода и други полезни практики.

Упражнения

За да затвърдим работата с методи, ще решим няколко задачи. В тях се изиска да напишете метод с определена функционалност и след това да го извикате като му подадете данни, прочетени от конзолата, точно както е показано в примерния вход и изход.

Задача: "Hello, Име!"

Да се напише метод, който получава като параметър име на човек и принтира на конзолата поздрав "Hello, <name>".

Примерен вход и изход

| Вход | Изход |
|-------|---------------|
| Peter | Hello, Peter! |

Насоки и подсказки

Дефинирайте метод **PrintName(string name)** и го имплементирайте, след което в главната програма прочетете от конзолата име на човек и извикайте метода като му подадете прочетеното име.

Тестване в Judge системата

Тествайте решението си: <https://judge.softuni.bg/Contests/Practice/Index/594#7>.

Задача: по-малко число

Да се създаде метод **GetMin(int a, int b)**, който връща по-малкото от две числа. Да се напише програма, която чете като входни данни от конзолата три числа и печата най-малкото от тях. Да се използва метода **GetMin(...)**, който е вече създаден.

Примерен вход и изход

| Вход | Изход | Вход | Изход |
|------|-------|------|-------|
| 1 | | -100 | |
| 2 | 1 | -101 | -102 |
| 3 | | -102 | |

Насоки и подсказки

Дефинирайте метод **GetMin(int a, int b)** и го имплементирайте, след което го извикайте от главната програма както е показано по-долу. За да намерите минимума на три числа, намерете първо минимума на първите две от тях и след това минимума на резултата и третото число:

```
var min = GetMin(GetMin(num1, num2), num3);
```

Тестване в Judge системата

Тествайте решението си: <https://judge.softuni.bg/Contests/Practice/Index/594#8>.

Задача: повтаряне на низ

Да се напише метод **RepeatString(str, count)**, който получава като параметри променлива от тип **string** и цяло число **n** и връща низа, повторен **n** пъти. След това резултатът да се отпечата на конзолата.

Примерен вход и изход

| Вход | Изход | Вход | Изход |
|----------|--------|-----------|------------------------------|
| str 2 | strstr | roki 6 | rokirokirokirokirokirokiroki |

Насоки и подсказки

Допишете метода по-долу като добавите съединяването на низове в цикъла:

```
static string RepeatString(string str, int count)
{
    string repeatedString = string.Empty;
    for (int i = 0; i < count; i++)
    {
        // TODO
    }
    return repeatedString;
}
```

Имайте предвид, че в езика C# съединяването на низове в цикъл води до лоша производителност и не се препоръчва. Потърсете и пробвайте по-ефективни решения тук: <https://stackoverflow.com/questions/411752>.

Тестване в Judge системата

Тествайте решението си: <https://judge.softuni.bg/Contests/Practice/Index/594#9>.

Задача: n-та цифра

Да се напише метод **FindNthDigit(number, index)**, който получава число и индекс N като параметри и печата N-тата цифра на числото (като се брои от дясно на ляво, започвайки от 1). След това, резултатът да се отпечата на конзолата.

Примерен вход и изход

| Вход | Изход |
|------------|-------|
| 83746 2 | 4 |

| Вход | Изход |
|---------------|-------|
| 93847837 6 | 8 |

| Вход | Изход |
|-----------|-------|
| 2435 4 | 2 |

Насоки и подсказки

За да изпълним алгоритъма, ще използваме **while** цикъл, докато дадено число не стане 0. На всяка итерация на **while** цикъла ще проверяваме дали настоящият индекс на цифрата не отговаря на индекса, който търсим. Ако отговаря, ще върнем като резултат цифрата на индекса (**number % 10**). Ако не отговаря, ще премахнем последната цифра на числото (**number = number / 10**). Трябва да следим коя цифра проверяваме по индекс (от дясно на ляво, започвайки от 1). Когато намерим цифрата, ще върнем индекса.

Тестване в Judge системата

Тествайте решението си: <https://judge.softuni.bg/Contests/Practice/Index/594#10>.

Задача: число към бройна система

Да се напише метод **IntegerToBase(number, toBase)**, който получава като параметри цяло число и основа на бройна система и връща входното число, конвертирано към посочената бройна система. След това, резултатът да се отпечата на конзолата. Входното число винаги ще е в бройна система 10, а параметърът за основа ще е между 2 и 10.

Примерен вход и изход

| Вход | Изход |
|--------|-------|
| 3 2 | 11 |

| Вход | Изход |
|--------|-------|
| 4 4 | 10 |

| Вход | Изход |
|--------|-------|
| 9 7 | 12 |

Насоки и подсказки

За да решим задачата, ще декларираме стрингова променлива, в която ще пазим резултата. След това трябва да изпълним следните изчисления, нужни за конвертиране на числото.

- Изчисляваме **остатъка** от числото, разделено на основата.
- **Вмъкваме** остатъка от числото в началото на низа, представящ резултата.
- **Разделяме** числото на основата.
- **Повтаряме** алгоритъма, докато входното число не стане 0.

Допишете липсващата логика в метода по-долу:

```

static string IntegerToBase(int number, int toBase)
{
    string result = "";
    while (number != 0)
    {
        // Implement the missing conversion logic
    }
    return result;
}

```

Тестване в Judge системата

Тествайте решението си: <https://judge.softuni.bg/Contests/Practice/Index/594#11>.

Задача: известия

Да се напише програма, която прочита цяло число **n** и на следващите редове въвежда **n** съобщения (като за всяко съобщение се прочитат по няколко реда). Всяко съобщение започва с **messageType**: **success**, **warning** или **error**:

- Когато **messageType** е **success** да се четат **operation** + **message** (всяко на отделен ред).
- Когато **messageType** е **warning** да се чете само **message**.
- Когато **messageType** е **error** да се четат **operation** + **message** + **errorCode** (всяко на отделен ред).

На конзолата да се отпечата всяко прочетено съобщение, форматирано в зависимост от неговия **messageType**. Като след заглавния ред за всяко съобщение да се отпечатат толкова на брой символа **=**, колкото е дълъг съответният заглавен ред и да се сложи по един празен ред след всяко съобщение (за по-детайлно разбиране погледнете примерите).

Задачата да се реши с дефинирането на четири метода: **ShowSuccessMessage()**, **ShowWarningMessage()**, **ShowErrorMessage()** и **ReadAndProcessMessage()**, като само последният метод да се извика от главния **Main()** метод:

```

static void ShowSuccessMessage(string operation,
    string message) { ... }

static void ShowWarningMessage(string message) { ... }

static void ShowErrorMessage(string operation,
    string message, int errorCode) { ... }

static void ReadAndProcessMessage() { ... }

```

Примерен вход и изход

| Вход | Изход |
|--|---|
| 4 error credit card purchase Invalid customer address | Error: Failed to execute credit card purchase. ===== |
| 500 warning Email not confirmed | Reason: Invalid customer address. Error code: 500. |
| success user registration User registered successfully warning Customer has not email assigned | Warning: Email not confirmed. ===== |
| | Successfully executed user registration. ===== |
| | User registered successfully. |
| | Warning: Customer has not email assigned. ===== |

Насоки и подсказки

Дефинирайте и имплементирайте посочените четири метода.

В **ReadAndProcessMessage()** прочетете типа съобщение от конзолата и според прочетения тип прочетете останалите данни (още един два или три реда). След това извикайте съответния метод за печатане на съответния тип съобщение.

Тестване в Judge системата

Тествайте решението си: <https://judge.softuni.bg/Contests/Practice/Index/594#12>.

Задача: числа към думи

Да се напише метод **Letterize(number)**, който прочита цяло число и го разпечатва с думи на английски език според условията по-долу:

- Да се отпечатат с думи стотиците, десетиците и единиците (и евентуални минус) според правилата на английския език.
- Ако числото е по-голямо от **999**, трябва да се принтира "too large".
- Ако числото е по-малко от **-999**, трябва да се принтира "too small".
- Ако числото е **отрицателно**, трябва да се принтира "**minus**" преди него.
- Ако числото не е съставено от три цифри, не трябва да се принтира.

Примерен вход и изход

| Вход | Изход |
|------|-------------------------------|
| 3 | nine-hundred and ninety nine |
| 999 | minus four-hundred and twenty |
| -420 | too large |
| 1020 | |

| Вход | Изход |
|------|-------------------------|
| 2 | fifteen |
| 15 | |
| 350 | three-hundred and fifty |

| Вход | Изход |
|-------|---------------------------|
| 4 | |
| 311 | three-hundred and eleven |
| 418 | four-hundred and eighteen |
| 509 | five-hundred and nine |
| -9945 | too small |

| Вход | Изход |
|------|------------------------------|
| 3 | |
| 500 | five-hundred |
| 123 | one-hundred and twenty three |
| 9 | nine |

Насоки и подсказки

Можем първо да отпечатаме стотиците като текст – (числото / 100) % 10, след тях десетиците – (числото / 10) % 10 и накрая единиците – (числото % 10).

Първият специален случай е когато числото е точно закръглено на 100 (напр. 100, 200, 300 и т.н.). В този случай отпечатваме "one-hundred", "two-hundred", "three-hundred" и т.н.

Вторият специален случай е когато числото, формирано от последните две цифри на входното число, е по-малко от 10 (напр. 101, 305, 609 и т.н.). В този случай отпечатваме "one-hundred and one", "three-hundred and five", "six-hundred and nine" и т.н.

Третият специален случай е когато числото, формирано от последните две цифри на входното число, е по-голямо от 10 и по-малко от 20 (напр. 111, 814, 919 и т.н.). В този случай отпечатваме "one-hundred and eleven", "eight-hundred and fourteen", "nine-hundred and nineteen" и т.н.

Тестване в Judge системата

Тествайте решението си: <https://judge.softuni.bg/Contests/Practice/Index/594#13>.

Задача: криптиране на низ

Да се напише метод **Encrypt(char letter)**, който криптира дадена буква по следния начин:

- Вземат се първата и последна цифра от ASCII кода на буквата и се залепят една за друга в низ, който ще представя резултата.
- Към началото на стойността на низа, който представя резултата, се залепя символа, който отговаря на следното условие:

- ASCII кода на буквата + последната цифра от ASCII кода на буквата.
- След това към края на стойността на низа, който представя резултата, се залепя символа, който отговаря на следното условие:
 - ASCII кода на буквата – първата цифра от ASCII кода на буквата.
 - Методът трябва да върне като резултат криптирания низ.

Пример:

- $j \rightarrow p16i$
- ASCII кодът на j е **106** → Първа цифра – **1**, последна цифра – **6**.
- Залепяме първата и последната цифра → **16**.
- Към **началото** на стойността на низа, който представя резултата, залепяме символа, който се получава от сума на ASCII кода + последната цифра → $106 + 6 \rightarrow 112 \rightarrow p$.
- Към **края** на стойността на низа, който представя резултата, залепяме символа, който се получава от разликата на ASCII кода – първата цифра → $106 - 1 \rightarrow 105 \rightarrow i$.

Използвайки метода, описан по-горе, да се напише програма, която чете **поредица от символи, криптира ги** и отпечатва резултата на един ред.

Примерен вход и изход

| Вход | Изход | Вход | Изход |
|--------------------------------------|------------------------------|-----------------------|------------------|
| 7 S o f t U n i | V83Kp11nh12ez16sZ85Mn10mn15h | 4 s I a r | x15rt18kh97Xr12o |

Приемаме, че входните данни винаги ще бъдат валидни. Главният метод трябва да прочита входните данни, подадени от потребителя – цяло число **n**, следвани от по един символ на всеки от следващите **n** реда.

Да се криптират символите и да се добавят към криптирания низ. Накрая като резултат трябва да се отпечата **криптиран низ от символи** като в примера:

- $S, o, f, t, U, n, i \rightarrow V83Kp11nh12ez16sZ85Mn10mn15h$

Насоки и подсказки

На променливата от тип **string**, в която ще се пази стойността на резултата, ще присвоим първоначална стойност **string.Empty**. Трябва да се завърти цикъл **n**

пъти, като на всяка итерация към променливата, в която пазим стойността на резултата, ще прибавяме криптирания символ.

За да намерим първата и последната цифри от ASCII кода, ще използваме алгоритъма, който използвахме за решаване на задача "Число към бройна система".

Тестване в Judge системата

Тествайте решението си: <https://judge.softuni.bg/Contests/Practice/Index/594#14>.

Глава 11. Хитрости и хакове

В настоящата глава ще разгледаме някои хитрости, хакове и техники, които ще улеснят работата ни с езика C# в среда за разработка Visual Studio. По-специално ще се запознаем:

- Как правилно да **форматираме код**
- С конвенции за **именуване на елементи от код**
- С някои **бързи клавиши** (keyboard shortcuts)
- С някои **шаблони с код** (code snippets)
- С техники за **дебъгване на код**

Форматиране на кода

Правилното форматиране на нашия код ще го направи **по-четим и разбираем**, в случай че се наложи някой друг да работи с него. Това е важно, защото в практиката ще ни се наложи да работим в екип с други хора и е от голямо значение дали пишем кода си така, че колегите ни да могат **бързо да се ориентират** в него.

Има определени правила за правилно форматиране на кода, които събрани в едно се наричат **конвенции**. Конвенциите са група от правила, общоприети от програмистите на даден език, и се ползват масово. Тези конвенции помагат за изграждането на норми в дадени езици – как е най-добре да се пише и какви са добрите практики. Приема се, че ако един програмист ги спазва, то кодът му е лесно четим и разбираем.

Езикът C# е направен от **Microsoft** и те са тези, които определят най-добрите практики за писане. Трябва да знаете също така, че дори да не спазвате конвенциите, наложени от **Microsoft**, кодът ви ще работи (стига да е написан правилно), но просто няма да бъде лесно разбираем. Това, разбира се, не е фатално на основно ниво, но колкото по-бързо свикнете да пишете качествен код, толкова по-добре.

Официалната **C# код конвенция** на Microsoft е публикувана в статията "C# Coding Conventions" в MSDN (<https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/inside-a-program/coding-conventions>) и в тази книга ще се ръководим основно от нея.

За форматиране на кода от Microsoft се препоръчва **къдрявите скоби {}** да са на отделен ред и точно под конструкцията, към която се отнасят, както е в примера по-долу.

```
if (true)
{
    Console.WriteLine("Inside the if statement");
}
```

Вижда се, че командалата **Console.WriteLine(...)** в примера е **4 празни полета на-вътре** (един таб), което също се препоръчва от **Microsoft**.

Ако дадена конструкция с къдрави скоби е един таб навътре, то **къдравите скоби {}** трябва да са в **началото на конструкцията**, както е в примера по-долу:

```
if (someCondition)
{
    if (anotherCondition)
    {
        Console.WriteLine("Inside the if statement");
    }
}
```

Ето това е пример за **лошо форматиран код** спрямо общоприетите конвенции за писане на код на езика C#:

```
if(true){
    Console.WriteLine("Inside the if statement");}
```

Първото, което се забелязва са **къдравите скоби {}**. Първата (отваряща) скоба трябва да е **точно под if условието**, а втората (затваряща) скоба – **под командата Console.WriteLine(...)**, на **отделен празен ред**. В допълнение, командата вътре в **if** конструкцията трябва да бъде **4 празни полета навътре (един таб)**. Веднага след ключовата дума **if** и преди условието на проверката се оставя **интервал**.

Същото правило важи и за **for цикли** и **всякакви други конструкции с къдрави скоби {}**. Ето още няколко примера:

Правилно:

```
for (int i = 0; i < 5; i++)
{
    Console.WriteLine(i);
}
```

Грешно:

```
for(int i=0;i<5;i++){
    Console.WriteLine(i);
}
```

За ваше удобство има **бързи клавиши** във **Visual Studio**, за които ще обясним по-късно в настоящата глава, но засега ни интересуват 2 конкретни комбинации. Едната комбинация е за форматиране на **кода в целия документ**, а другата комбинация – за форматиране на **част от кода**. Ако искаме да форматираме **целия код**, то трябва да натиснем **[CTRL + K + D]**. В случай, че искаме да форматираме само **част от кода**, то ние трябва да **маркираме с мишката частта**, която искаме да форматираме, и да натиснем **[CTRL + K + F]**.

Нека използваме **грешния пример** от преди малко:

```
for(int i=0;i<5;i++){
    Console.WriteLine(i);
}
```

Ако натиснем [CTRL + K + D], което е нашата комбинация за форматиране на **целия документ**, ще получим код, форматиран според **общоприетите конвенции за C#**, който ще изглежда по следния начин:

```
for (int i = 0; i < 5; i++)
{
    Console.WriteLine(i);
}
```

Тази комбинация може да ни помогне, ако попаднем на лошо форматиран код.

Именуване на елементите на кода

В тази секция ще се фокусираме върху **общоприетите конвенции за именуване на проекти, файлове и променливи**, наложени от Microsoft.

Именуване на проекти и файлове

За именуване на **проекти и файлове** се препоръчва описателно име, което подсказва каква е ролята на въпросния файл / проект и в същото време се препоръчва **PascalCase** конвенцията. Накратко, това е **конвенция за именуване** на елементи, при която всяка дума, включително първата, започва с **главна буква**, например **ExpressionCalculator**.

Пример: в този курс се започва с лекция на име **First steps in coding** и следователно едно примерно именуване на проекта (solution) за тази лекция може да бъде **FirstStepsInCoding**. Същата конвенция важи и за файловете в даден проект. Ако вземем за пример първата задача от лекцията **First steps in coding**, тя се казва **Hello World** и следователно нашият файл в проекта ще се казва **HelloWorld**.

Именуване на променливи

В програмирането променливите пазят някакви данни и за да е по-разбираем кода, името на една променлива трябва **да подсказва нейното предназначение**. Ето и още няколко препоръки за имената на променливите:

- Името трябва да е **кратко и описателно** и да обяснява за какво служи дадената променлива.
- Името трябва да се състои само от буквите **a-z, A-Z**, цифрите **0-9**, както и символа **'_'**.
- В C# е прието променливите да **започват** винаги с **малка буква** и да **съдържат малки букви**, като **всяка следваща дума** в тях започва с **главна буква** (това именуване е още познато като **camelCase** конвенция).

- Трябва да се внимава за главни и малки букви, тъй като C# прави разлика между тях. Например **age** и **Age** са различни променливи.
- Имената на променливите не могат да съвпадат със служебна дума (keyword) от езика C#, например **int** е невалидно име на променлива.



Въпреки че използването на символа `_` в имената на променливите е разрешено, в C# това не се препоръчва и се счита за лош стил на именуване.

Ето няколко примера за **добре именувани** променливи:

- **firstName**
- **age**
- **startIndex**
- **lastNegativeNumberIndex**
- **linesCount**

Ето няколко примера за **лошо именувани променливи**, макар и имената да са коректни от гледна точка на компилатора на C#:

- **_firstName** (започва с '`_`')
- **last_name** (съдържа '`_`')
- **AGE** (изписана е с главни букви)
- **Start_Index** (започва с главна буква и съдържа '`_`')
- **lastNegativeNumber_Index** (съдържа '`_`')

Първоначално всички тези правила може да ни се струват безсмислени и ненужни, но с течение на времето и натрупването на опит ще видите нуждата от норми за писане на качествен код, за да може да се работи по-лесно и по-бързо в екип. Ще разберете, че е изключително досадна работата с код, който е написан без да се спазват никакви правила за качествен код.

Бързи клавиши във Visual Studio

В предната секция споменахме за две от комбинациите, които се отнасят за форматиране на код. Едната комбинация [CTRL + K + D] беше за **форматиране на целия код в даден файл**, а втората [CTRL + K + F] ни служеше в случай, че искаме да **форматираме само дадена част от кода**. Тези комбинации се наричат **бързи клавиши** и сега ще дадем по-подробна информация за тях.

Бързи клавиши са **комбинации**, които ни предоставят възможността да извършваме някои действия **по-лесно и по-бързо**, като всяка среда за разработка на софтуер си има своите бързи клавиши, въпреки че повечето се повтарят. Сега ще разгледаме някои от **бързите клавиши** във **Visual Studio**.

| Комбинация | Действие |
|--------------------|--|
| [CTRL + F] | Комбинацията отваря търсачка , с която можем да търсим в нашия код. |
| [CTRL + K + C] | Закоментира част от кода. |
| [CTRL + K + U] | Разкоментира код, който е вече закоментиран. |
| [CTRL + Z] | Връща една промяна назад (т.нр. Undo). |
| [CTRL + Y] | Комбинацията има противоположно действие на [CTRL + Z] (т.нр. Redo). |
| [CTRL + K + D] | Форматира кода според конвенциите по подразбиране. |
| [CTRL + Backspace] | Изтрива думата вляво от курсора. |
| [CTRL + Del] | Изтрива думата вдясно от курсора. |
| [CTRL + Shift + S] | Запазва всички файлове в проекта. |
| [CTRL + S] | Запазва текущия файл. |

Повече за **бързите клавиши във Visual Studio** може да намерите тук: <https://shortcutworld.com/en/Visual-Studio/2015/win/all>.

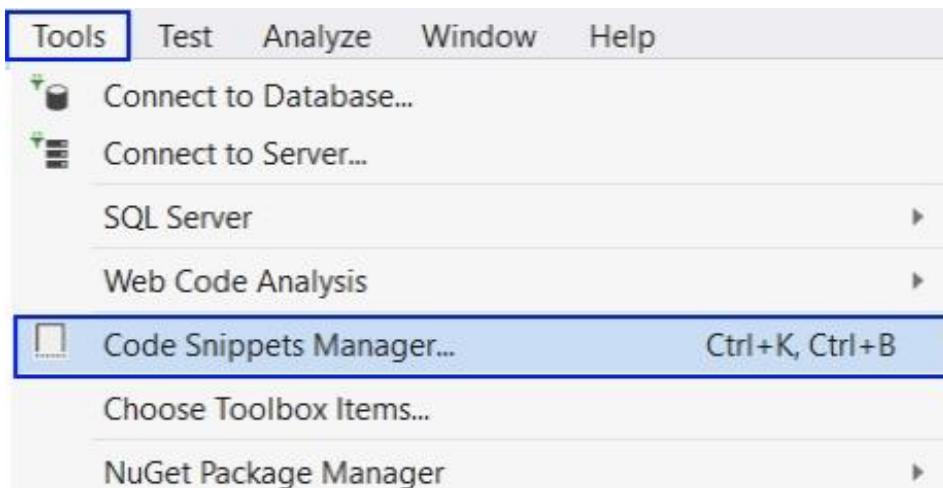
Шаблони с код (code snippets)

Във Visual Studio съществуват т.нр. **шаблони с код** (code snippets), при изписването на които се генерира по шаблон някакъв блок с код. Примерно, при изписването на кратък код “**cw**” и **натискане на [Tab] + [Tab]** се генерира кодът **Console.WriteLine();** в тялото на нашата програма, на мястото на краткия код. Това се нарича “разгъване на шаблон за кратък код”. Подобно работи и шаблона **“for” + [Tab] + [Tab]**. На фигурата по-долу е показано действието на шаблона “**cw**”:

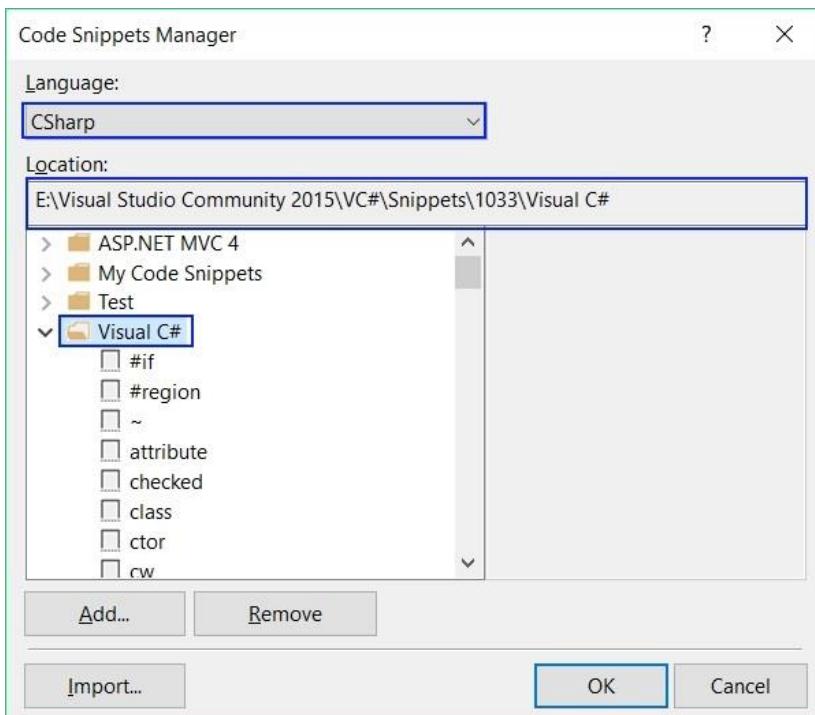


Да си направим собствен шаблон за код

В тази секция ще покажем как сами да си направим собствен шаблон. Ще разгледаме как се прави code snippet за **Console.ReadLine()**. Като за начало ще си създадем нов празен проект и ще отидем на [Tools -> Code Snippets Manager], както е показано на снимката:



В отворилия се прозорец трябва да изберем **Language -> CSharp**, а от секцията **Locations -> Visual C#**. Там се намират всички съществуващи шаблони за езика C#:



Избираме някой snippet, например **cw**, вземаме пътя до неговия файл и го отваряме с редактора на Visual Studio:

```

1  <?xml version="1.0" encoding="utf-8" ?>
2  <CodeSnippets xmlns="http://schemas.microsoft.com/VisualStudio/2005/CodeSnippet">
3  <CodeSnippet Format="1.0.0">
4  <Header>
5    <Title>cw</Title>
6    <Shortcut>cw</Shortcut>
7    <Description>Code snippet for Console.WriteLine</Description>
8    <Author>Microsoft Corporation</Author>
9    <SnippetTypes>
10      <SnippetType>Expansion</SnippetType>
11    </SnippetTypes>
12  </Header>
13  <Snippet>
14    <Declarations>
15      <Literal Editable="false">
16        <ID>SystemConsole</ID>
17        <Function>SimpleTypeName(global::System.Console)</Function>
18      </Literal>
19    </Declarations>
20    <Code Language="csharp"><![CDATA[$SystemConsole$.WriteLine($end$);]]>
21    </Code>
22  </Snippet>
23 </CodeSnippet>
24 </CodeSnippets>

```

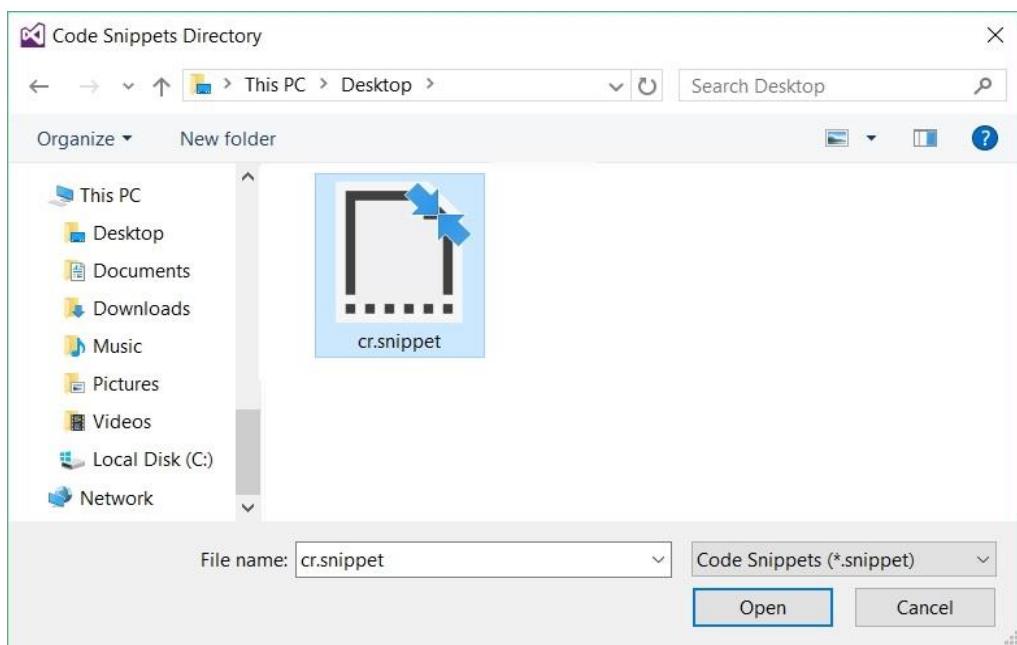
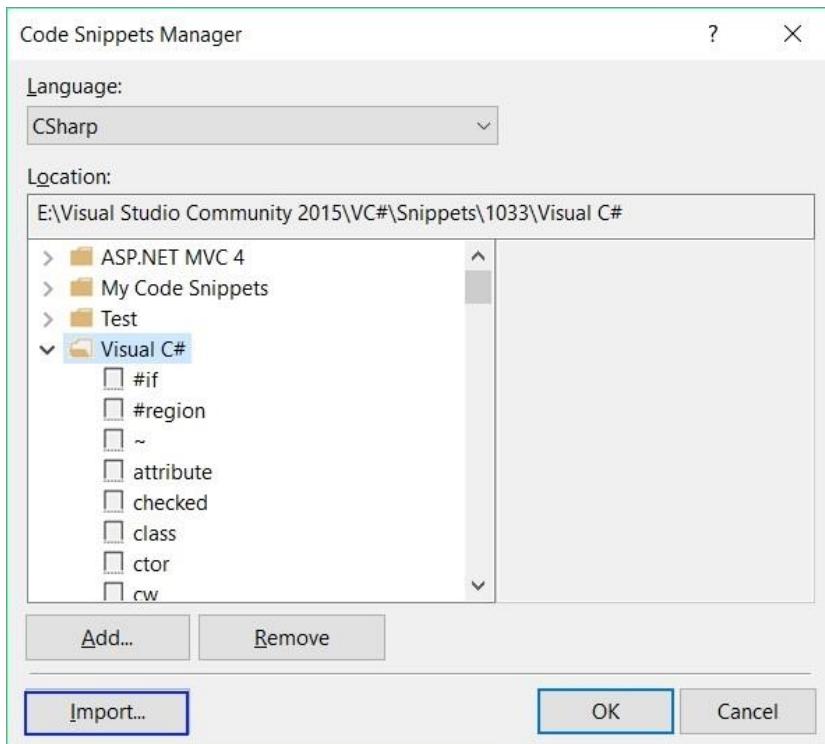
Виждаме доста непознати неща, но няма страшно, по-нататък ще се запознаем и с тях. Сега се фокусираме върху частта **<Title><Title>**, **<Shortcut><Shortcut>** и кода между **CDATA[]**. Първо ще сменим заглавието, което седи в секцията **<Title><Title>** и вместо **cw**, ще напишем **cr**, като това ще бъде **заглавието на нашия шаблон**. След това, в секцията **<Shortcut><Shortcut>**, ще сменим това, което трябва да напишем за **извикването на нашия шаблон** (shortcut) от **cw** на **cr**. Накрая трябва да сменим кода в **CDATA[]**, от **WriteLine** на **ReadLine**: **CDATA[\$SystemConsole\$.ReadLine(\$end\$);]**. Пожелание може да промените и секциите **Description** и **Author**. Промененият файл трябва да изглежда така:

```

1  <?xml version="1.0" encoding="utf-8" ?>
2  <CodeSnippets xmlns="http://schemas.microsoft.com/VisualStudio/2005/CodeSnippet">
3  <CodeSnippet Format="1.0.0">
4  <Header>
5    <Title>cr</Title>
6    <Shortcut>cr</Shortcut>
7    <Description>Code snippet for Console.ReadLine</Description>
8    <Author>SoftUni Wizard</Author>
9    <SnippetTypes>
10      <SnippetType>Expansion</SnippetType>
11    </SnippetTypes>
12  </Header>
13  <Snippet>
14    <Declarations>
15      <Literal Editable="false">
16        <ID>SystemConsole</ID>
17        <Function>SimpleTypeName(global::System.Console)</Function>
18      </Literal>
19    </Declarations>
20    <Code Language="csharp"><![CDATA[$SystemConsole$.ReadLine($end$);]]>
21    </Code>
22  </Snippet>
23 </CodeSnippet>
24 </CodeSnippets>

```

След като сме написали нашия snippet, трябва да си **запазим** файла във формат **snippetName.snippet** (в нашия случай **cr.snippet**) и да го добавим към Visual Studio. Отиваме в [Tools] -> [Code Snippet Manager] -> [Import] и избираме **cr.snippet** файла, който създадохме:



Вече когато напишем **cr** във Visual Studio, **нашият нов snippet** се появява:



Техники за дебъгване на кода

Дебъгването играе важна роля в процеса на създаване на софтуер, която ни позволява **постъпково да проследим изпълнението** на нашата програма. С помощта на тази техника можем да **следим стойностите на локалните променливи**, тъй като те се променят по време на изпълнение на програмата, и да **отстраним евентуални грешки** (бъгове). Процесът на дебъгване включва:

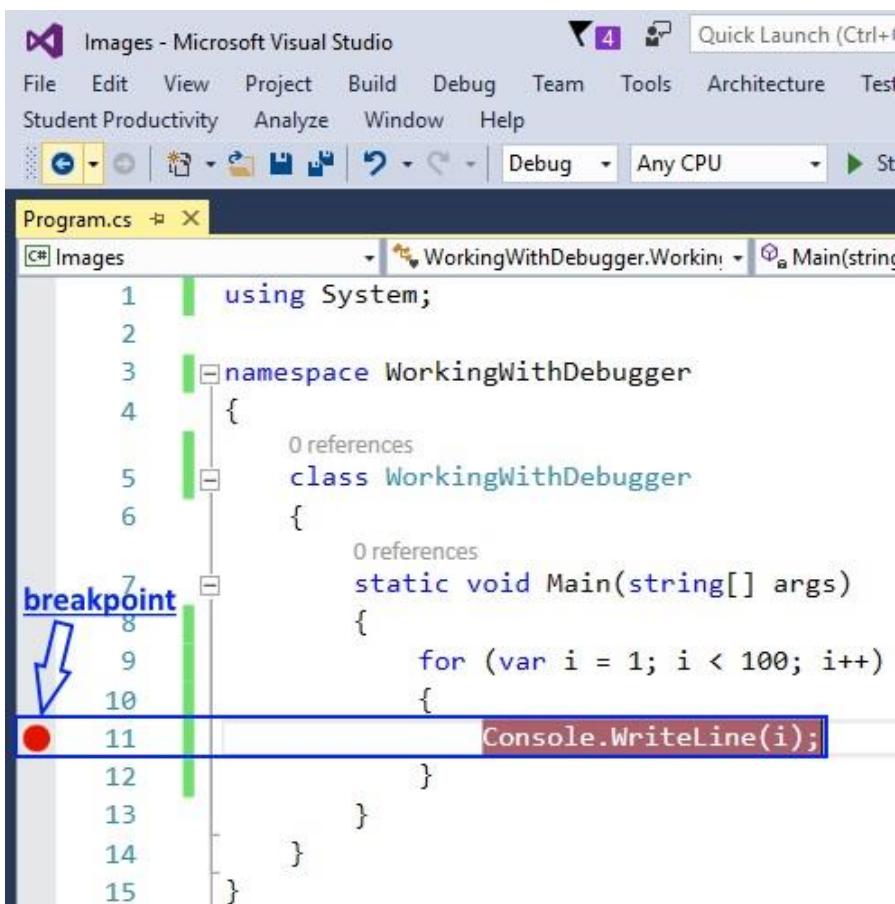
- **Забелязване** на проблемите (бъговете).
- **Намиране** на кода, който причинява проблемите.
- **Коригиране** на кода, причиняващ проблемите, така че програмата да работи правилно.
- **Тестване**, за да се убедим, че програмата работи правилно след нанесените корекции.

Visual Studio ни предоставя **вграден дебъгер** (debugger), чрез който можем да поставяме **точки на прекъсване** (или breakpoints), на избрани от нас места. При среща на **стопер** (breakpoint), програмата **спира изпълнението** си и позволява **постъпково изпълнение** на останалите редове. Дебъгването ни дава възможност да **вникнем в детайлите на програмата** и да видим къде точно възникват грешките и каква е причината за това.

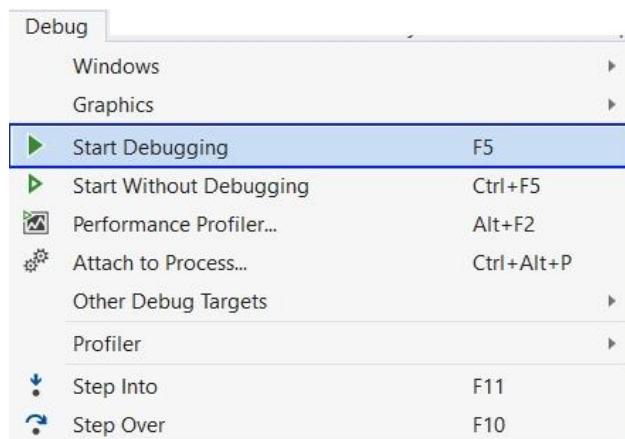
За да демонстрираме работа с дебъгера ще използваме следната програма:

```
static void Main(string[] args)
{
    for (int i = 0; i < 100; i++)
    {
        Console.WriteLine(i);
    }
}
```

Ще сложим **стопер** (breakpoint) на функцията **Console.WriteLine(...)**. За целта трябва да преместим курсора на реда, който печата на конзолата, и да натиснем **[F9]**. Появява се **стопер**, където програмата ще **спре** изпълнението си:

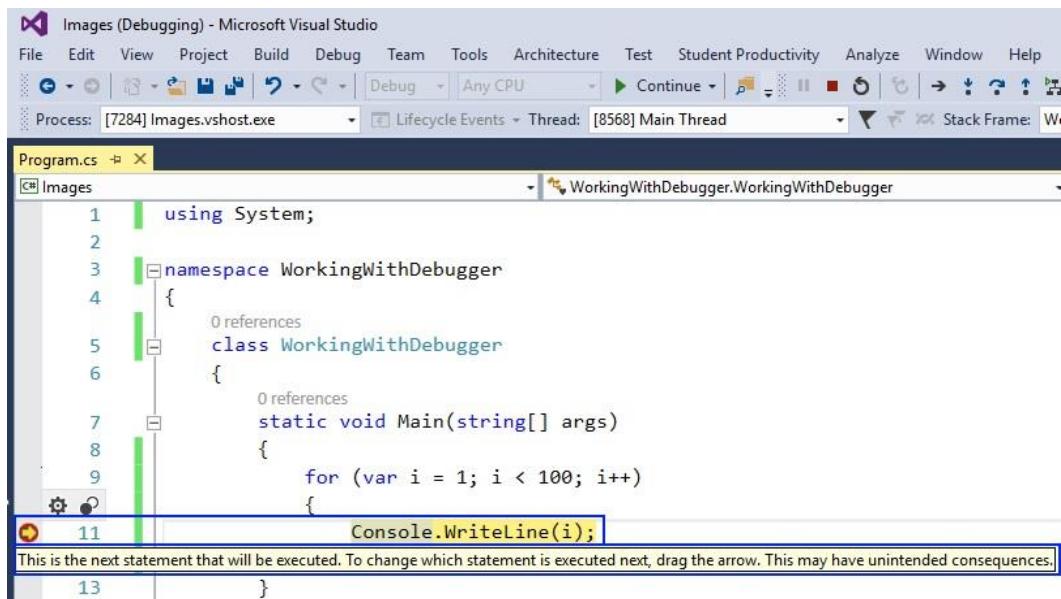


За да стартираме програмата в режим на дебъгване, избираме [Debug] -> [Start Debugging] или натискаме [F5]:



След стартиране на програмата виждаме, че тя спира изпълнението си на ред 11, където сложихме стопера (breakpoint). Кодът на текущия ред се оцветява с жълт цвят и можем да го изпълняваме постъпково. За да преминем на следващ ред

използваме клавиши [F10]. Забелязваме, че кодът на текущия ред все още не е изпълнен. Изпълнява се, когато преминем на следващия ред:



От прозореца **Locals** можем да наблюдаваме **промените по локалните променливи**. За да отворите прозореца изберете [Debug] -> [Windows] -> [Locals].

| Locals | | |
|--------|-------------|----------|
| Name | Value | Type |
| args | {string[0]} | string[] |
| i | 4 | int |

Справочник с хитрости

В тази секция ще припомним накратко **хитрости и техники** от програмирането с езика C#, разглеждани вече в тази книга, които ще са ви много полезни, ако ходите на изпит по програмиране за начинаещи:

Вкарване на променливи в стринг (string)

```
var text = "some text";
Console.WriteLine("{0}", text);
// Това ще отпечата на конзолата "some text"
```

В случая използваме **placeholder** – **{x}**, където **x** е число (по-голямо или равно на 0), отговарящо на позицията, на която трябва да поставим променливата. Следователно, ако подаваме две променливи ще имаме един placeholder, който ще е **{0}** и той ще съдържа стойността на **първата променлива** и още един – **{1}**, който ще съдържа стойността на **втората променлива**. Примерно:

```
var text = "some text";
var number = 5;
Console.WriteLine("{0} {1} {0}", text, number);
// Това ще отпечата на конзолата "some text 5 some text"
```

В този пример забелязваме, че можем да подаваме **не само текстови променливи**. Също така можем да използваме дадена променлива **няколко пъти** и за целта в placeholder-а слагаме числото, което **отговаря на позицията на променливата**. В случая на нулева позиция стои променливата **text**, а на първа позиция е променливата **number**. Номерацията е обикновена отначало, но трябва да запомните, че в програмирането **броенето започва от 0**.

Форматиране с 2 цифри след десетичния знак

```
var number = 5.432432;
Console.WriteLine(Math.Round(number, 2));
// Това ще отпечата на конзолата "5.43"
```

Math.Round(...) приема 2 параметъра:

- първият е **числото, което искаме да закръглим**
- вторият е **числото, което определя с колко символа след десетичния знак ще закръглим** (това число винаги трябва да бъде цяло число)

Ако искаме да закръглим до **2 цифри след десетичния знак** и третата цифра е по-малка от 5, както в примера по-горе, то закръглянето е надолу, но ако третата цифра е 5 или по-голяма – закръглянето е нагоре, както е в примера по-долу:

```
var number = 5.439;
Console.WriteLine(Math.Round(number, 2));
// Това ще отпечата на конзолата "5.44"
```

Други методи за закръгляне

В случай, че искаме винаги да закръгляме надолу можем вместо **Math.Round(...)** да използваме друг метод – **Math.Floor(...)**, който винаги закръгля надолу, но също така винаги закръгля до цяло число. Например, ако имаме числото 5.99 и използваме **Math.Floor(5.99)**, ще получим числото **5**.

Можем и да направим точно обратното – винаги да закръгляме нагоре, използвайки метода **Math.Ceiling(...)**. Отново, ако имаме число примерно 5.11 и използваме **Math.Ceiling(5.11)**, ще получим 6. Ето и няколко примера:

```
var numberToFloor = 5.99;
Console.WriteLine(Math.Floor(numberToFloor));
// Това ще отпечата на конзолата 5
```

```
var numberToCeiling = 5.11;
Console.WriteLine(Math.Ceiling(numberToCeling));
// Това ще отпечата на конзолата 6
```

Закръгляне чрез placeholder

```
var num = 5.432424;
Console.WriteLine("{0:f2}", num);
```

В случая след числото добавяме **:f2**, което ще ограничи числото до 2 цифри след десетичния знак и ще работи като **Math.Round(...)**. Трябва да имаме предвид, че числото след буквата **f** означава до колко цифри след десетичния знак да е закръглено числото (т.е. може да е примерно **f3** или **f5**).

Как се пише условна конструкция?

Условната **if** конструкция се състои от следните елементи:

- Ключова дума **if**
- Булев израз (условие)
- Тяло на условната конструкция
- Незадължително: **else** клауза

```
if (условие)
{
    // тяло
}
else (условие)
{
    // тяло
}
```

За улеснение може да използваме code snippet за **if** конструкция:

- “**if**” + [Tab] + [Tab]

Как се пише for-цикъл?

За **for** цикъл ни трябват няколко неща:

- Инициализационен блок, в който се декларира променливата-брояч (**var i**) и се задава нейна начална стойност.
- Условие за повторение (**i <= 10**).
- Обновяване на брояча (**i++**).

- Тяло на цикъла.

```
for (var i = 0; i < 5; i++)  
{  
    // тяло  
}
```

За улеснение може да използваме **code snippet** за **for** цикъл:

- “**for**” + [Tab] + [Tab]

Какво научихме от тази глава?

В настоящата глава се запознахме как **правилно** да **форматираме** и именуваме елементите на нашия **код**, някои **бързи клавиши** (shortcuts) за работа във Visual Studio, **шаблони с код** (code snippets) и разгледахме как се **дебъгва код**.

Заключение

Ако сте прочели цялата книга и сте решили всички задачи от упражненията и сте стигнали до настоящото заключение, заслужавате **поздравления!** Вече сте направили **първата стъпка** от изучаването на професията на програмиста, но имате още доста **дълъг път** докато станете **истински добри** и превърнете **писането на софтуер** в своя професия.

Спомнете си за [четирите основни групи умения](#), които всеки програмист трябва да притежава, за да работи своята професия:

- Умение #1 – **писане на програмен код** (20% от уменията на програмиста) – покриват се до голяма степен от тази книга, но трябва да изучите още базови структури от данни, класове, обекти, функции, стрингове и други елементи от писането на код.
- Умение #2 – **алгоритмично мислене** (30% от уменията на програмиста) – покриват се частично от тази книга и се развиват най-вече с решаване на голямо количество разнообразни алгоритмични задачи.
- Умение #3 – **фундаментални знания за професията** (25% от уменията на програмиста) – усвояват се за няколко години с комбинация от учене и практикуване (четене на книги, гледане на видео уроци, посещаване на курсове и най-вече писане на разнообразни проекти от различни технологични области).
- Умение #4 – **езици за програмиране и софтуерни технологии** (25% от уменията на програмиста) – усвояват се продължително време, с много практика, здраво четене и писане на проекти. Тези знания и умения бързо оставят и трябва непрестанно да се актуализират. Добрите програмисти учат всеки ден нови технологии.

Тази книга е само първа стъпка!

Настоящата книга по основи на програмирането е само **първа стъпка** от изграждането на уменията на един програмист. Ако сте успели да решите **всички задачи**, това означава, че сте **получили ценни знания** за принципите на програмиране с езика C# на **базисно ниво**. Тепърва ви предстои да изучавате **по-задълбочено** програмирането, както и да развивате **алгоритмичното си мислене**, след което да добавите и **технологични знания** за езика C# и .NET екосистемата (.NET Framework, .NET Core, Entity Framework, ASP.NET и други), front-end технологиите (HTML, CSS, JavaScript) и още редица концепции, технологии и инструменти за разработка на софтуер.

Ако **не сте успели** да решите всички задачи или голяма част от тях, върнете се и ги решете! Помните, че за да **станете програмисти** се изискват **много труд и усилия**. Тази професия не е за мързеливци. Без **да практикувате сериозно** програмирането години наред, няма как да го научите!

Както вече обяснихме, първото и най-базово умение на програмиста е да се научи да пише код с лекота и удоволствие. Именно това е мисията на тази книга: да ви научи да кодите. Препоръчваме ви освен книгата, да запишете и **практическия курс "Основи на програмирането"** в СофтУни (<https://softuni.bg/apply>), който се предлага напълно безплатно в присъствена или онлайн форма на обучение.

Накъде да продължим след тази книга?

С тази книга сте поставили стабилни основи, благодарение на които ще ви е лесно да продължите да се развивате като програмисти. Ако се чудите как да продължите развитието си, помислете за следните няколко възможности:

- да учите за **софтуерен инженер** в СофтУни и да направите програмирането своя професия;
- да продължите развитието си като програмист **по свой собствен път**, например чрез самообучение или с някакви онлайн уроци;
- да си **останете на ниво кодер**, без да се занимавате с програмиране по-сериозно.

Професия "софтуерен инженер" в СофтУни

Първата, и съответно препоръчителната, възможност да овладеете цялостно и на ниво професията "софтуерен инженер", е да започнете своето обучение по **цялостната програма на СофтУни за подготовка на софтуерни инженери**: <https://softuni.bg/curriculum>. Учебният план на СофтУни е внимателно разработен от **д-р Светлин Наков и неговия екип**, за да ви поднесе последователно и с градираща сложност всички умения, които един софтуерен инженер трябва да притежава, за да стартира кариера като разработчик на софтуер в ИТ фирма.

Продължителност на обучението в СофтУни

Обучението в СофтУни е с продължителност **2-3 години** (в зависимост от професията и избраните специализации) и за това време е нормално да достигнете добро начално ниво (junior developer), но **само ако учите сериозно** и здраво пишете код всеки ден. При добър успех един типичен студент **започва работа на средата на обучението си** (след около 1.5 години). Благодарение на добре развита партньорска мрежа **кариерният център на СофтУни предлага работа** в софтуерна или ИТ фирма на всички студенти в СофтУни, които имат много добър или отличен успех.

Програмист се става за най-малко година здраво писане на код

Предупреждаваме ви, че **програмист се става с много усилия**, с писане на десетки хиляди редове код и с решаване на стотици, дори хиляди практически задачи, и отнема години! Ако някой ви предлага "по-лека програма" и ви обещава да станете програмисти и да започнете работа за 3-4 месеца, значи или ви **лъже**,

или ще ви даде толкова ниско ниво, че **няма да ви вземат даже за стажант**, дори и да си плащате на фирмата, която си губи времето с вас. Има и изключения, разбира се, например ако не започвате от нулата или ако имате екстремно развито инженерно мислене или ако кандидатствате за много ниска позиция (например техническа поддръжка), но като цяло **програмист за по-малко от 1 година здраво учене и писане на код не се става!**

Приемен изпит в СофтУни

За **да се запишете в СофтУни** е нужно да се явите на **приемен изпит** по "Основи на програмирането" върху материала от тази книга. Ако решавате с лекота задачите от упражненията в книгата, значи сте готови за изпита. Обърнете внимание и на няколкото глави за **подготовка за практически изпит по програмиране**. Те ще ви дадат добра представа за трудността на изпита и за типовете задачи, които трябва да се научите да решавате.

Ако задачите от книгата и подготовкителните примерни изпити ви затрудняват, значи имате **нужда от още подготовка**. Запишете се на безплатния курс по "Основи на програмирането" (<https://softuni.bg/apply>) или преминете внимателно през книгата още веднъж отначало, без да пропускате да решавате **задачите от всяка една учебна тема!** Трябва да се научите **да ги решавате с лекота**, без да си помагате с насоките и примерните решения.

Учебен план за софтуерни инженери

След изпита ви очаква **сериозен учебен план** по програмата на СофтУни за обучение на софтуерни инженери. Той е поредица от **модули с по няколко курса** по програмиране и софтуерни технологии, изцяло насочени към усвояване на фундаментални познания от разработката на софтуер и придобиване на **практически умения за работа** като програмист с най-съвременните софтуерни технологии. На студентите се предоставя избор между **няколко професии** и специализации с фокус върху C#, Java, JavaScript, PHP и други езици и технологии. Всяка професия се изучава в няколко модула с продължителност от 4 месеца и всеки модул съдържа 2 или 3 курса. Учебните занятия са разделени на **теоретична подготовка (30%) и практически упражнения, проекти и занимания (70%)**, а всеки курс завършва с практически изпит или практически курсов проект.

Колко часа на ден отнема обучението?

Обучението за софтуерен инженер в СофтУни е **много сериозно занимание** и трябва да му отделите като **минимум поне по 4-5 часа всеки ден**, а за препоръчване е да посветите цялото си време на него. Съчетанието на **работка с учене** невинаги е успешно, но ако работите нещо леко с много свободно време, е добър вариант. СофтУни е подходяща възможност за **ученици, студенти и работещи други професии**, но най-добре е да отделите цялото си време за вашето образование и овладяването на професията. Не става с 2 или 4 часа на седмица!

Формите на обучение в СофтУни са присъствена (по-добър избор) и онлайн (ако нямате друга възможност). И в двете форми, за да успеете да научите предвиденото в учебния план (което се изисква от софтуерните фирми за започване на работа), е необходимо здраво учене. Просто трябва да намерите време! Причина #1 за буксуване по пътя към професията в СофтУни е неотделянето на достатъчно време за обучението: като минимум поне 20-30 часа на седмица.

Софтуни за работещи и учащи другаде

На всички, които изкарат отличен резултат на приемния изпит в СофтУни и се запалят истински по програмирането и мечтаят да го направят своя професия, препоръчваме да се освободят от останалите си ангажименти и да отделят цялото си време, за да научат професията "софтуерен инженер" и да започнат да си изкарват хляба с нея.

- За **работещите** това означава да си напуснат работата (и да вземат заем или да си свият финансовите разходи, за да изкарат с по-нисък доход 1-2 години до започване на работа по новата професия).
- За **учащите** в традиционен университет това означава да си изместят силно фокуса към програмирането и практическите курсове в СофтУни, като намалят до минимум времето, което отделят за традиционния университет.
- За **безработните** това е отличен шанс да вложат цялото си време, сили и енергия, за да придобият една перспективна, добре платена и много търсена професия, която ще им осигури високо качество на живот и дългосрочен просперитет.
- За **учениците** от средните училища и гимназиите това означава да си сложат приоритет какво е по-важно за тяхното развитие: да учат практическо програмиране в СофтУни, което ще им даде професия и работа или да отделят цялото си внимание на традиционната образователна система или да съчетават умело и двете начинания. За съжаление, често пъти приоритетите се задават от родителите и за тези случаи нямаме решение.

На всички, които не могат да изкарат отличен резултат на приемния изпит в СофтУни препоръчваме да набледнат върху по-доброто изучаване, разбиране и най-вече практикуване на учебния материал от настоящата книга. Ако не се справяте с лекота със задачите от тази книга, няма да се справяте и за напред при изучаването на програмирането и разработката на софтуер.

Не пропускайте основите на програмирането! В никакъв случай не трябва да взимате смели решения да напускате работата си или традиционния университет и да кроите велики планове за бъдещата си професия на софтуерен инженер, ако нямате отличен резултат на входния изпит в СофтУни! Той е мерило доколко ви се отдава програмирането, доколко ви харесва и доколко наистина сте мотивирани да го учите сериозно и да го работите след това години наред всеки ден с желание и наслада.

Професия "софтуерен инженер" по ваш собствен път

Другата възможност за развитие след тази книга е да продължите да изучавате програмирането извън СофтУни. Можете да запишете или да следите **видео курсове**, които навлизат в по-голяма дълбочина в програмирането със C# или други езици и платформи за разработка. Можете да четете книги за програмиране и софтуерни технологии, да следвате **онлайн ръководства (tutorials)** и други онлайн ресурси – има безкрайно много бесплатни материали в Интернет. Запомнете, обаче, че най-важното по пътя към професията на програмиста е да правите практически проекти!

Без писане на много, много код и здраво практикуване, не се става програмист. Отделете си **достатъчно време**. Програмист не се става за месец или два. В Интернет ще намерите голям набор от **свободни ресурси** като книги, учебници, видео уроци, онлайн и присъствени курсове за програмиране и разработка на софтуер. Обаче, ще трябва да инвестирате **поне година-две**, за да добиете начално ниво като за започване на работа.

След като понапреднете, намерете начин или да започнете **стаж в някоя фирма** (което ще е почти невъзможно без поне година здраво писане на код преди това) или да си измислите **ваш собствен практически проект**, по който да поработите няколко месеца, даже година, за да се учате чрез проба-грешка.



Запомнете, че има много начини да станете програмисти, но всички те имат обща пресечна точка: **здраво писане на код и практика години наред!**

Онлайн общности за стартиращите в програмирането

Независимо по кой път сте поели, ако ще се занимавате сериозно с програмиране, е препоръчително да следите специализирани **онлайн форуми, дискусионни групи и общности**, от които можете да получите помощ от свои колеги и да следите новостите от софтуерната индустрия.

Ако ще учате програмиране сериозно, **обградете се с хора**, които се занимават с програмиране сериозно. Присъединете се към **общности от софтуерни разработчици**, ходете по софтуерни конференции, ходете на събития за програмисти, намерете си приятели, с които да си говорите за програмиране и да си обсъждате проблемите и бъговете, намерете среда, която да ви помага. В София и в големите градове има бесплатни събития за програмисти, по няколко на седмица. В по-малките градове имате Интернет и достъп до цялата онлайн общност.

Ето и някои препоръчителни **ресурси** за развитието ви като програмист:

- softuni.bg – официален **уеб сайт на СофтУни**. В него ще намерите бесплатни (и не само) курсове, семинари, видео уроци и обучения по програмиране, софтуерни технологии и дигитални компетенции.

- softuni.bg/forum – официален **форум на СофтУни**. Форумът за дискусии на СофтУни е изключително позитивен и изпълнен с желаещи да помогат колеги. Ако зададете смислен въпрос, свързан с програмирането и изучаваните в СофтУни технологии, почти сигурно ще получите смислен отговор до минути. Опитайте, нищо не губите.
- [fb.com/SoftwareUniversity](https://www.facebook.com/SoftwareUniversity) – официална **Facebook страница на СофтУни**. От нея ще научавате за нови курсове, семинари и събития, свързани с програмирането и разработката на софтуер.
- introprogramming.info – официален уеб сайт на **книгите "Въведение в програмирането"** със **C#** и **Java** от д-р Светлин Наков и колектив. Книгите разглеждат в дълбочина основите на програмирането, базовите структури от данни и алгоритми, ООП и други базови умения и са отлично продължение за четене след настоящата книга. Обаче **освен четене, трябва и здраво писане**, не забравяйте това!
- stackoverflow.com – **Stack Overflow** е един от **най-големите** в световен мащаб дискусионни форуми за програмисти, в който ще получите помощ за всеки възможен въпрос от света на програмирането. Ако владеете английски език, търсете в StackOverflow и задавайте въпросите си там.
- [fb.com/groups/bg.developers](https://www.facebook.com/groups/bg.developers) – групата "**Програмиране България**" е една от най-големите онлайн Facebook общности от програмисти за дискусии по темите на софтуерната разработка на български език.
- meetup.com/find/tech/ – потърсете **технологични срещи** (tech meetups) около вас и се включете в общностите, които харесвате. Повечето технологични срещи са безплатни и новобранци са добре дошли.
- Ако се интересувате от ИТ събития, технологични конференции, обучения и стажове, разгледайте и по-големите **сайтове за ИТ събития** като dev.bg и iteventz.bg.

Успех на всички!

От името на целия авторски колектив ви **пожелаваме неспирни успехи в професията и в живота!** Ще сме невероятно щастливи, ако с наша помощ сте се **запалили по програмирането** и сме ви вдъхновили да поемете смело към професията "софтуерен инженер", която ще ви донесе добра работа, която да работите с удоволствие, качествен живот и просперитет, като и страхотни перспективи за развитие и възможности да правите значими проекти с вдъхновение и страсть.

София, 21 май 2017 г.



C#

В ръцете си държите нещо повече от **книга за програмиране**, учебник или учебно помагало. Това съвременно образователно пособие ви повежда по **първите стъпки към програмирането** чрез малко текст и **много код**, наситено с примери и внимателно подбрани **практически задачи** със система за моментално **автоматично оценяване**.

Учебното съдържание е разработено лично от **д-р Светлин Наков**, който през 15-годишния си опит с обучението на софтуерни инженери помага на **над 70 000 души** да навлязат в програмирането и намира как да поднася информацията на **малки порции**, с много практика и с **нарастваща сложност**.

Запомнете, че програмиране се учи с **много писане на код и усилено решаване на задачи** и не става само с четене на книги, така че преминете старательно през **упражненията**. Успех!

Сайт: csharp-book.softuni.bg



Авторски колектив:

Александър Кръстев
Александър Лазаров
Ангел Димитриев
Васко Викторов
Венцислав Петров
Даниел Цветков
Димитър Татарски
Димо Димов
Диян Тончев
Елена Роглева
Живко Недялков
Жулиета Атанасова
Захария Пехливанова
Ивелин Кирилов
Искра Николова
Калин Примов
Кристиян Памидов
Любослав Любенов
Николай Банкин
Николай Димов
Павлин Петков
Петър Иванов
Росица Ненова
Руслан Филипов
Светлин Наков
Стефка Василева
Теодор Куртев
Тоньо Желев
Християн Христов
Христо Христов
Цветан Илиев
Юlian Линев
Яница Вълева

ISBN 978-619-00-0635-0



9 786190 006350 >