

**СВЕТЛИН НАКОВ**  
**& КОЛЕКТИВ**

# JavaScript

**ОСНОВИ НА  
ПРОГРАМИРАНЕТО С  
JavaScript**





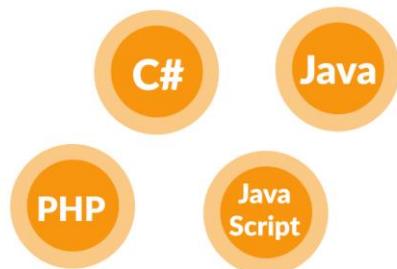
# Кратко съдържание

Кратко съдържание .....	3
Съдържание .....	7
Предговор .....	13
Глава 1. Първи стъпки в програмирането.....	27
Глава 2.1. Прости пресмятания с числа.....	55
Глава 2.2. Прости пресмятания с числа – изпитни задачи.....	83
Глава 3.1. Прости проверки.....	99
Глава 3.2. Прости проверки – изпитни задачи .....	127
Глава 4.1. По-сложни проверки .....	141
Глава 4.2. По-сложни проверки – изпитни задачи .....	171
Глава 5.1. Повторения (цикли) .....	191
Глава 5.2. Повторения (цикли) – изпитни задачи .....	215
Глава 6.1. Вложени цикли.....	233
Глава 6.2. Вложени цикли – изпитни задачи .....	253
Глава 7.1. По-сложни цикли.....	267
Глава 7.2. По-сложни цикли – изпитни задачи .....	287
Глава 8.1. Подготовка за практически изпит – част I .....	299
Глава 8.2. Подготовка за практически изпит – част II .....	325
Глава 9.1. Задачи за шампиони – част I.....	341
Глава 9.2. Задачи за шампиони – част II.....	357
Глава 10. Функции.....	373
Глава 11. Хитрости и хакове .....	401
Заключение.....	415

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](http://softuni.bg/apply)

# Основи на програмирането с JavaScript

Светлин Наков и колектив

Бончо Вълков

Венцислав Петров

Димитър Далев

Елена Роглева

Жулиета Атанасова

Захария Пехливанова

Здравко Костадинов

Ивелин Арнаудов

Кристиан Мариянов

Мартин Чаов

Николай Банкин

Николай Костов

Павел Колев

Петър Иванов

Светлин Наков

Стилян Канголов

Християн Христов

Христо Минков

ISBN: 978-619-00-0702-9

София, 2018

# Основи на програмирането с JavaScript

© Светлин Наков и колектив, 2018 г.

Първо издание: май 2018 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 опишете 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: 978-619-00-0702-9

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

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

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

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

# Съдържание

Кратко съдържание .....	3
Съдържание .....	7
Предговор .....	13
За кого е тази книга? .....	13
Защо избрахме езика JavaScript? .....	14
Книгата на други програмни езици: C#, Java, Python, C++, PHP .....	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 .....	23
Официален сайт на книгата .....	24
Форум за вашите въпроси .....	24
Официална Facebook страница на книгата .....	25
Лиценз и разпространение .....	25
Докладване на грешки .....	25
Приятно четене! .....	25
<b>Глава 1. Първи стъпки в програмирането.....</b>	<b>27</b>
Видео .....	27
Какво означава "да програмираме"? .....	27
Как да напишем компютърна програма? .....	33
Среда за разработка (IDE) .....	34
Пример: конзолна програма "Hello JavaScript" .....	37
Изпълняване на код в браузър чрез HTML + JS .....	42
Типични грешки в JavaScript програмите .....	43
Какво научихме от тази глава? .....	44
Упражнения: първи стъпки в коденето .....	44
Конзолни, графични и уеб приложения .....	50
Упражнения: уеб приложения .....	50
<b>Глава 2.1. Прости пресмятания с числа.....</b>	<b>55</b>
Видео .....	55
Пресмятания в програмирането .....	55
Типове данни и променливи .....	55
Печатане на резултат на екрана .....	56
Четене на потребителски вход – цяло число .....	56
Четене на дробно число .....	58
Четене на вход – текст .....	59
Съединяване на текст и числа .....	59
Аритметични операции .....	60

Съединяване на текст и число .....	61
Числени изрази.....	62
Закръгляне на числа .....	63
Какво научихме от тази глава? .....	65
Упражнения: прости пресмятания.....	65
Графични приложения с числови изрази .....	75
<b>Глава 2.2. Прости пресмятания с числа – изпитни задачи .....</b>	<b>83</b>
Преговор: четене, извеждане на числа и пресмятания.....	83
Изпитни задачи .....	84
Задача: учебна зала.....	84
Задача: зеленчукова борса .....	87
Задача: ремонт на плочки .....	90
Задача: парички .....	93
Задача: дневна печалба .....	96
<b>Глава 3.1. Прости проверки .....</b>	<b>99</b>
Видео.....	99
Оператори за сравнение .....	99
Прости проверки.....	100
Проверки с if-else конструкция .....	101
За къдравите скоби { } след if / else.....	102
Живот на променлива .....	104
Серии от проверки.....	105
Упражнения: прости проверки .....	106
Дебъгване - прости операции с дебъгер .....	111
Упражнения: прости проверки .....	112
Графично уеб приложение .....	122
<b>Глава 3.2. Прости проверки – изпитни задачи .....</b>	<b>127</b>
Изпитни задачи .....	127
Задача: цена за транспорт .....	127
Задача: тръби в басейн.....	130
Задача: поспалившата котка Том .....	132
Задача: реколта .....	135
Задача: фирма .....	137
<b>Глава 4.1. По-сложни проверки.....</b>	<b>141</b>
Видео.....	141
Вложени проверки .....	141
По-сложни проверки.....	144
Логическо "ИЛИ" .....	146
Логическо отрицание.....	148

---

Операторът скоби () .....	148
По-сложни логически условия.....	148
Условна конструкция switch-case.....	152
Какво научихме от тази глава?.....	155
Упражнения: по-сложни проверки .....	156
Упражнение: графично приложение с по-сложни проверки.....	161
<b>Глава 4.2. По-сложни проверки – изпитни задачи.....</b>	<b>171</b>
Вложени проверки.....	171
Switch-case проверки .....	171
Изпитни задачи .....	172
Задача: навреме за изпит .....	172
Задача: пътешествие.....	176
Задача: операции между числа.....	180
Задача: билети за мач.....	183
Задача: хотелска стая.....	187
<b>Глава 5.1. Повторения (цикли).....</b>	<b>191</b>
Видео .....	191
Повторения на блокове код (for цикъл).....	191
Code Snippet за for цикъл във Visual Studio Code .....	192
Какво научихме от тази глава? .....	200
Упражнения: повторения (цикли) .....	201
Упражнения: графични и уеб приложения .....	204
<b>Глава 5.2. Повторения (цикли) – изпитни задачи .....</b>	<b>215</b>
Изпитни задачи .....	215
Задача: хистограма .....	215
Задача: умната Лили .....	219
Задача: завръщане в миналото .....	222
Задача: болница .....	224
Задача: деление без остатък .....	227
Задача: логистика .....	229
<b>Глава 6.1. Вложени цикли.....</b>	<b>233</b>
Видео .....	233
Вложени цикли .....	234
Чертане на по-сложни фигури .....	240
Какво научихме от тази глава? .....	247
Упражнения: чертане на фигурки в уеб среда.....	247
<b>Глава 6.2. Вложени цикли – изпитни задачи .....</b>	<b>253</b>
Изпитни задачи .....	253
Задача: чертане на крепост .....	253

Задача: пеперуда.....	256
Задача: знак "Стоп" .....	258
Задача: стрелка.....	260
Задача: брадва .....	263
<b>Глава 7.1. По-сложни цикли.....</b>	<b>267</b>
Видео.....	267
Цикли със стъпка.....	267
While цикъл.....	270
Най-голям общ делител (НОД) .....	272
Алгоритъм на Евклид.....	272
Do-while цикъл .....	273
Безкрайни цикли и операторът break.....	275
Вложени цикли и операторът break.....	279
Задачи с цикли.....	281
Какво научихме от тази глава? .....	286
<b>Глава 7.2. По-сложни цикли – изпитни задачи .....</b>	<b>287</b>
Изпитни задачи .....	287
Задача: генератор за тъпи пароли .....	287
Задача: магически числа .....	289
Задача: спиращо число .....	293
Задача: специални числа .....	295
Задача: цифри .....	296
<b>Глава 8.1. Подготовка за практически изпит – част I.....</b>	<b>299</b>
Видео.....	299
Практически изпит по “Основи на програмирането” .....	299
Система за онлайн оценяване (Judge).....	299
Задачи с прости пресмятания.....	299
Задачи с единична проверка .....	303
Задачи с по-сложни проверки .....	307
Задачи с единичен цикъл .....	311
Задачи за “чертане” на фигурки на конзолата .....	315
Задачи с вложени цикли с по-сложна логика .....	319
<b>Глава 8.2. Подготовка за практически изпит – част II.....</b>	<b>325</b>
Изпитни задачи .....	325
Задача: разстояние .....	325
Задача: смяна на плочки .....	328
Задача: магазин за цветя.....	330
Задача: оценки .....	333
Задача: коледна шапка .....	335

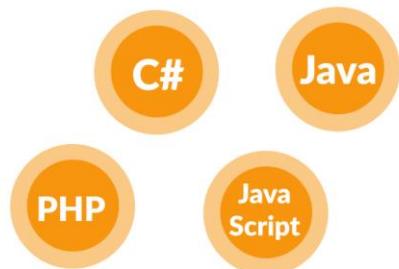
---

Задача: комбинации от букви.....	337
<b>Глава 9.1. Задачи за шампиони – част I.....</b>	<b>341</b>
По-сложни задачи върху изучавания материал.....	341
Задача: пресичащи се редици .....	341
Задача: магически дати .....	345
Задача: пет специални букви .....	350
<b>Глава 9.2. Задачи за шампиони – част II.....</b>	<b>357</b>
По-сложни задачи върху изучавания материал.....	357
Задача: дни за страстно пазаруване.....	357
Задача: числен израз .....	362
Задача: бикове и крави .....	366
<b>Глава 10. Функции.....</b>	<b>373</b>
Какво е "функция"? .....	373
Деклариране на функции .....	374
Извикване на функции .....	376
Функции с параметри .....	378
Връщане на резултат от функция .....	383
Варианти на функции.....	388
Вложени функции .....	388
Именуване на функции. Добри практики при работа с функции .....	389
Какво научихме от тази глава?.....	392
Упражнения .....	392
<b>Глава 11. Хитрости и хакове .....</b>	<b>401</b>
Форматиране на кода .....	401
Именуване на променливи .....	403
Бързи клавиши във Visual Studio Code .....	404
Шаблони с код (code snippets).....	406
Техники за дебъгване на кода .....	409
Справочник с хитрости .....	410
Какво научихме от тази глава? .....	413
<b>Заключение.....</b>	<b>415</b>
Тази книга е само първа стъпка!.....	415
Накъде да продължим след тази книга? .....	416
Онлайн общности за стартиращите в програмирането .....	419
Успех на всички! .....	420

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](http://softuni.bg/apply)

# Предговор

Книгата "Основи на програмирането" е официален учебник за курса "Programming Basics" по програмиране за начинаещи 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**) и работа с функции (деклариране и извикване на функции, подаване на параметри и връщане на стойност). Използват се езикът за програмиране **JavaScript** и средата за разработка **Visual Studio Code**. 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 българските професионални 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](#), 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**, помощ 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 всякаква друга помощ за начинаещия.

Бесплатният 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 стапиращи в програмирането е достъпно през **формата за кандидатстване в СофтУни**: <https://softuni.bg/apply>.

## Защо избрахме езика JavaScript?

За настоящата книга избрахме езика **JavaScript**, защото е **съвременен език** за програмиране от високо ниво и същевременно е лесен за научаване и **подходящ за начинаещи**. Като употреба **JavaScript** е **широкоразпространен**, с добре развита екосистема, с многобройни библиотеки и технологични рамки и съответно дава много **перспективи** за развитие.

**JavaScript** комбинира парадигмите на процедурното, функционалното и обектно-ориентираното програмиране по съвременен начин с лесен за употреба синтаксис. В книгата ще използваме **езика JavaScript** и средата за разработка **Visual Studio Code**, която е достъпна бесплатно от Microsoft.

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

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

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

- [Основи на програмирането със C#](#)
- [Основи на програмирането с Java](#)
- [Основи на програмирането с JavaScript](#)
- [Основи на програмирането с Python](#)
- [Основи на програмирането със C++](#)
- [Основи на програмирането с PHP](#)

Ако предпочитате друг език, 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 здраво задачи**, 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 тества. Още по-важни от примерите, обаче, 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 повечето 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 (**Софтуни**)

Софтуерният университет (**Софтуни**) е най-мащабният учебен център за софтуерни инженери в България. През него преминават десетки хиляди студенти всяка година. Софтуни отваря врати през 2014 г. като продължение на усилията на д-р Светлин Наков масирано да изгражда **kadъrni softuerni speciaлисти** 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

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 запознят с **основните програмни конструкции** от света на софтуерната разработка, които ще можете да приложите при всеки един език за програмиране. Те включват работа с **данни, променливи и изрази**, използване на **проверки**, конструиране на **цикли** и дефиниране и извикване на **функции** 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** съпътства 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**: пращате решение и след секунди получавате 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 системата** 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 системата:

- 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: <https://softuni.bg>.
- Използването на системата е **бесплатно** и не е обвързано с участието в курсовете на Софтуни.

Убедени сме, че след няколко изпратени задачи, **ще ви хареса да получавате моментална обратна връзка** дали написаното от вас решение е вярно, и Judge системата 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 пригответи 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, 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 уменията, които **всеки добър програмист притежава** и към които всеки новобранец трябва да се стреми:

- писане на код (20%)
- алгоритмично мислене (30%)
- фундаментални знания за професията (25%)
- езици и технологии за разработка (25%)

## Умение #1 – кодене (20%)

Да се научите **да пишете код** формира около 20% от минималните умения на програмиста, необходими за започване на работа в софтуерна фирма. 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 до съвършенство 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.

Специално обръщаме внимание, че **езикът за програмиране няма съществено значение** за умението да кодиш. Или можеш да кодиш или не. Ако можеш да кодиш на **JavaScript**, 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](#)) 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 **структури от данни** (масиви, списъци, матрици, хеш-таблици, дърводидни структури) и базови **алгоритми** (търсене, сортиране, обхождане на дърводидни структури, рекурсия).

**Умения за алгоритмично мислене** се развиват сериозно в началните курсове за софтуерни инженери в СофтУни (вж. [учебния план](#)), както и в специализираните курсове по [структурите от данни](#) и [алгоритми](#).

Както може би се досещате, **езикът за програмиране няма значение** за развитието 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%)

Фундаменталните 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 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
- **умения да програмираш** - писане на код, работа с данни, ползване на условни конструкции и цикли, работа с масиви, списъци и асоциативни масиви, стрингове и текстообработка, работа с потоци и файлове, ползване на програмни интерфейси (APIs), работа с дебъгер и други
- **структури от данни и алгоритми** - списъци, дървета, хеш-таблици, графи, търсене, сортиране, рекурсия, обхождане на дърводидни структури и други
- **обектно-ориентирано програмиране** (ООП) – работа с класове, обекти, наследяване, полиморфизъм, абстракция, интерфейси, капсуляция на данни, 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, frameworks за уеб разработка, templating engines
- **уеб front-end технологии (клиентска разработка)** - HTML, CSS, JS, HTTP, DOM, AJAX, комуникация с back-end, извикване на REST API, front-end frameworks, базов дизайн и UX (user experience) концепции
- **мобилни технологии** - мобилни приложения, Android и iOS разработка, мобилен потребителски интерфейс (UI), извикване на сървърна логика
- **вградени системи** - микроконтролери, управление на цифров и аналогов вход и изход, достъп до сензори, управление на периферия
- **операционни системи** - работа с операционни системи (Linux, Windows и други), инсталация, конфигурация и базова системна администрация, работа с процеси, памет, файлова система, потребители, многозадачност, виртуализация и контейнери
- **паралелно програмиране и асинхронност** - управление на нишки, асинхронни задачи, promises, общи ресурси и синхронизация на достъпа
- **софтуерно инженерство** - сорс контрол системи, управление на разработката, планиране и управление на задачи, 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 в 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](#), както и с редица [изборни курсове](#). 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 научаване, но най-бързо се променят с времето. Ако погледнем **обявите за работа** от софтуерната индустрия, там често се споменават всякачи думички (като изброените по-долу), но всъщност в обявите мълчаливо **се подразбираят първите три умения**: да кодиш, да мислиш алгоритично 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 към май 2018 г.):

- **JavaScript** (JS) + ООП + ФП + бази данни + MongoDB или MySQL + HTTP + уеб програмиране + HTML + CSS + DOM + jQuery + Node.js + Express + Angular или React
- **C#** + ООП + ФП + класовете от .NET + база данни SQL Server + Entity Framework (EF) + 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
- **Python** + ООП + ФП + бази данни + MongoDB или 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 + 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 алгоритмично (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 почти няма съществено значение**, просто трябва да се научите да програмирате, а това започва с **коденето** (настоящата книга), продължава в по-сложните **концепции от програмирането** (като структури от данни, алгоритми, ООП и ФП) и включва усвояването 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 **JavaScript**, но той не е съществен и може да се замени с Java, C#, 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 алгоритмично** 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 **бесплатни 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.

Бесплатните обучения на СофтУни за старт в програмирането са може би най-мащабните, провеждани някога в България. До 2018 г. курсът на СофтУни по основи на програмирането **се провежда над 170 пъти в близо 40 български града** присъствено и многократно онлайн, с над 60 000 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 усилията на [Фондация "Софтуерен университет"](#) 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):

Бончо Вълков, Венцислав Петров, Димитър Далев, Елена Роглева, Жулиета Атанасова, Захария Пехливанова, Здравко Костадинов, Ивелин Арнаудов, Кристиан Мариянов, Мартин Чаов, Николай Банкин, Николай Костов, Павел Колев, Петър Иванов, Светлин Наков, Стилян Кангалов, Християн Христов, Христо Минков

Книгата е базирана на нейния първоначален C# вариант ([Въведение в програмирането със C#](#)), който е разработен от широк авторски колектив и който има принос и към настоящата книга:

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 с JavaScript** за начинаещи е достъпна за свободно ползване в Интернет от адрес:

<https://js-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 (като C#, Java, Python и C++), 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 безплатно **адекватен отговор** по **всякакви въпроси** от учебното **съдържание на настоящия учебник**, както и по 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 наследствано 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 умивате от няколко часа, **задайте го във форума** и ще получите отговор. Ще се убедите 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 **безплатно** 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 receptionта на СофтУни (вж. <https://softuni.bg/contacts>).

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

Международен стандартен номер на книга ISBN: 978-619-00-0702-9.

## 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-JS-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.

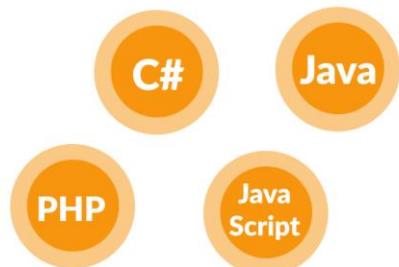
## Приятно четене!

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



**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](http://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** (Integrated Development Environment - накратко IDE) и как да работим с тях, в частност с **Visual Studio Code**. Ще напишем и изпълним **първата си програма** на програмния език **JavaScript**, а след това ще се упражним с няколко задачи: ще създадем конзолна програма и уеб приложение. Ще се научим как да проверяваме за коректност решенията 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 видеоурок по тази глава тук: <https://youtu.be/0YkrJsKwHdM>.

## 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 **JavaScript**, **Python**, **C#**, **Java**, **PHP**, **Ruby**, **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 **JavaScript**. Затова в настоящата книга, ще се запознаем със синтаксиса и семантиката на езика **JavaScript** и с програмирането като цяло, 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 Code.

## Езици за програмиране, компилатори, интерпретатори и среда 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 и междувременно **проверява за грешки в 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 и 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 JavaScript** много често се ползва средата за разработка **Visual Studio Code**, която се разработва и разпространява безплатно от Microsoft и може да се изтегли от: <https://www.visualstudio.com/downloads>. Алтернативи на Visual Studio Code са **WebStorm** (<https://www.jetbrains.com/webstorm>), **Atom** (<https://atom.io>) и други. В настоящата книга ще използваме средата за разработка Visual Studio Code.

## Езици от ниско и високо ниво, среди за изпълнение (Runtime Environments)

Програмата в своята същност е **набор от инструкции**, които карат компютъра да свърши определена задача. Те се въвеждат от програмиста и се **изпълняват безусловно 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), уеббраузъри, компилатори, двигатели за графика и игри (game engines) и други системни компоненти и програми. С езици от още по-високо ниво като **JavaScript**, **Python** и **C#** 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 **JavaScript**, **Python**, **C#**, **Java**, **C++**, **PHP** и др.

**JavaScript** е **интерпретиран език**, а това означава, че пишем команди, които се изпълняват директно след стартиране на програмата. Това означава, че ако сме допуснали грешка при писането на код, ще разберем едва след стартиране на програмата и достигането до грешната команда. Тук на помощ идват **IDE-тата**, като **Visual Studio Code**, които проверяват кода ни, още докато пишем и ни алармират за евентуални проблеми. Когато сме написали кода си и искаем да го тестваме, можем да го запаметим във файл с разширение **.js**.

Повечето **езици за програмиране** са специализирани в разработването на конкретен вид приложения - Desktop - за Windows или Mac, мобилни приложения, сървърни приложения, приложения за управление на умни джаджи и т.н. **JavaScript** е един от малкото езици, които позволяват да създадете

приложение във всяка една възможна област - от сайтове и мобилни приложения до сървърни и desktop приложения.

Най-популярните среди за интерпретация на JavaScript са **уеб браузърите**, които използвате всеки ден - Chrome, Firefox, Internet Explorer и т.н. Когато зареждате любимият си уеб сайт е много вероятно той да съдържа **JavaScript файлове**, които ще се изпълнят при отварянето на сайта и ще направят изживяването при разглеждането на сайта по - приятно и динамично. Твърде вероятно е ако в сайта има падащи менюта, анимации, регистрация на потребител, дразнещи реклами, то те да са реализирани именно чрез използването на езика **JavaScript**.

Често ще чувате че даден код се "изпълнява на клиента" - това ще рече, че **JavaScript** кодът, се изпълнява във вашият **browser**, който играе ролята на клиент, или приемник. За да има приемник, трябва да има и предавател. В технологичният свят тези предаватели се наричат сървъри. Можете да си представяте сървърите като едни много мощни компютри, до които имат достъп много хора. Всички сайтове стоят на подобни сървъри - т.е. файловете които помагат на сайта да изглежда такъв какъвто го виждате - **картинки, текстове, JavaScript файлове**, се намират на конкретен сървър. Вашият **browser** (клиент) се свързва към **сървъра**, на който се намира сайта, който посещавате и сървърът **изпраща** обратно необходимите файлове, за да се визуализира пред вас любимият сайт. Можете да си представите, че вашият **browser** е **радиото** в колата ви на което слушате любимата програма, а сървърът е сградата от където се излъчва това предаване, макар и комуникацията между двете да протича по различен начин.

Другият популярен **интерпретатор за JavaScript** е **NodeJS**. Можете да си го представите като приложение, което инсталирате на компютъра си и той започва да разбира от **JavaScript** по същият начин, по който и вашият **browser** разбира. По този начин можете да изпълнявате **JavaScript** код **директно на компютъра си**, без необходимостта от **browser**. Както казахме сървърите са просто по - мощни компютри. Те започват да разбират от **JavaScript** по същият начин по който и вашият компютър - просто им се инсталлира **NodeJS**. Можете да си инсталirate **NodeJS** от официалния сайт <https://nodejs.org> напълно бесплатно, като следвате инструкциите.

## Компютърни програми - 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 запазва във файл с разширение **.js** (например **main.js**) след което може да се изпълни през вашия **уеб браузър** или през системата **конзолата** с помощта на **NodeJS**. След малко ще разгледаме и двета варианта.

## Компютърни програми – примери

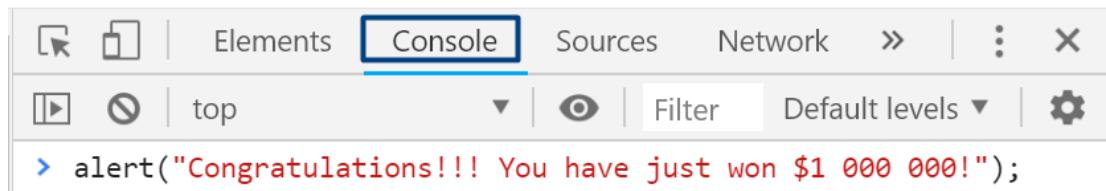
Да започнем с много прост пример за кратка **JavaScript програма**, която ще изпълним директно в уеб браузъра (той поддържа JS без да инсталираме нищо допълнително).

### Пример: програма, която нотифицира потребителя

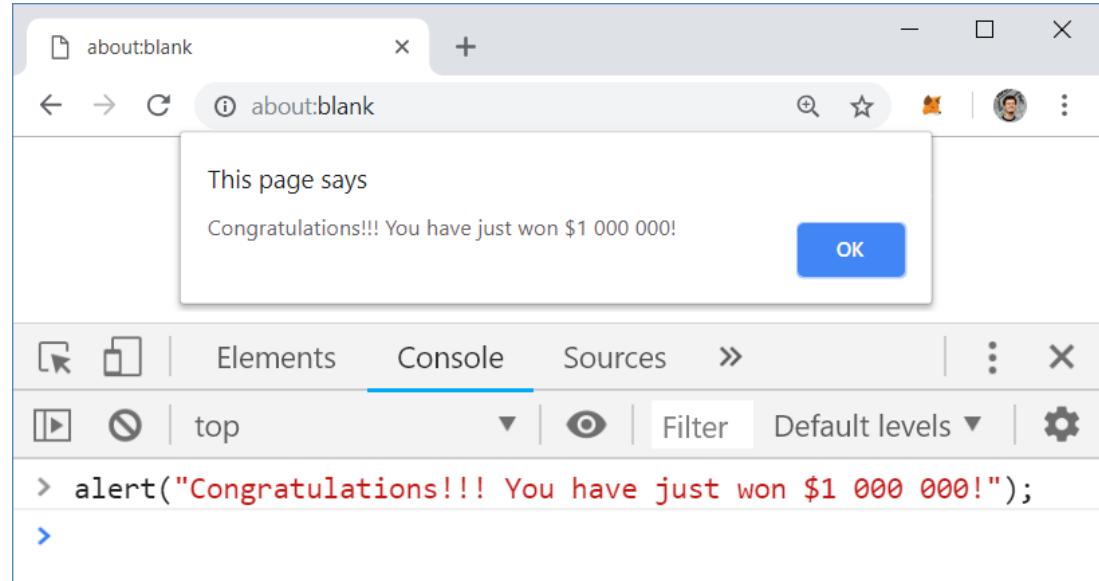
Нашата първа програма ще е единична **JavaScript команда**, която нотифицира потребителя, че е спечелил 1 000 000 долара, както често се случва, когато разглеждате някой сайт с много спам и реклами:

```
alert("Congratulations!!! You have just won $1 000 000!");
```

Можем да изпълним програмата като използваме JavaScript **конзолата в нашия уеб браузър**. Например в **Chrome** натискаме **[F12]** и пишем кода в прозорчето [**Console**]:



Резултатът е нещо такова: модално **popup съобщение** в браузъра:

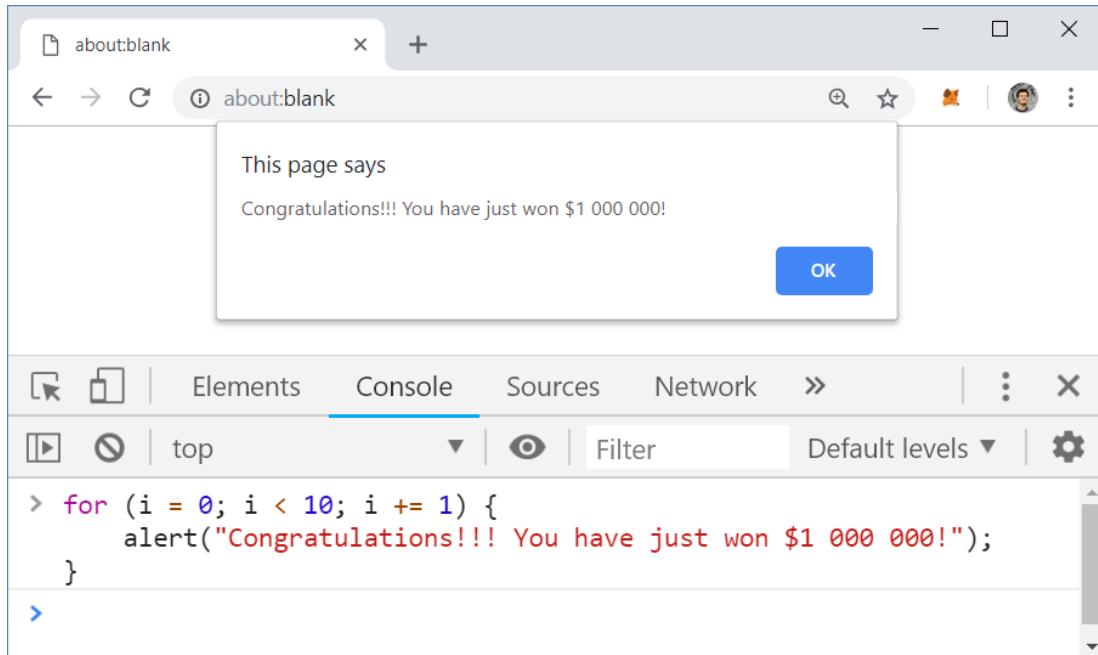


### Пример: програма, която спами нотификации на потребителя

Можем да усложним предходната програма, като зададем за изпълнение повтарящи се многократно в поредица команди за нотифициране на потребителя, че е победител в томбола:

```
for (i = 0; i < 10; i += 1) {
    alert("Congratulations!!! You have just won $1 000 000!");
}
```

В горният пример караме уеб браузъра да изкарва нотификации една след друга докато нотификациите не станат 10 на брой. **Резултатът** е един доста раздразнен потребител:



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 лева в евро

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

```
let myMoney = prompt("How much money do you want to convert:");

let leva = parseInt(myMoney);
let euro = leva / 1.95583;
console.log(euro);
```

Ако изпълним тази програма в JavaScript конзолата на браузъра, ще получим нещо такова:

The screenshot shows a browser window with a modal dialog asking for money to convert. The user has entered '50'. Below the modal is the developer tools' Console tab, which contains the following code and output:

```
> let myMoney = prompt("How much money do you want to convert:");

let leva = parseInt(myMoney);
let euro = leva / 1.95583;
console.log(euro);

25.564594059810926
VM264:5
< undefined
```

Разгледахме **три примера за компютърни програми**: единична команда, серия команди в цикъл и поредица от 4 команди. Нека сега преминем към по-интересното: как можем да пишем собствени програми на **JavaScript** и как можем да ги изпълняваме извън браузъра?

## Как да напишем компютърна 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 **JavaScript** 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.

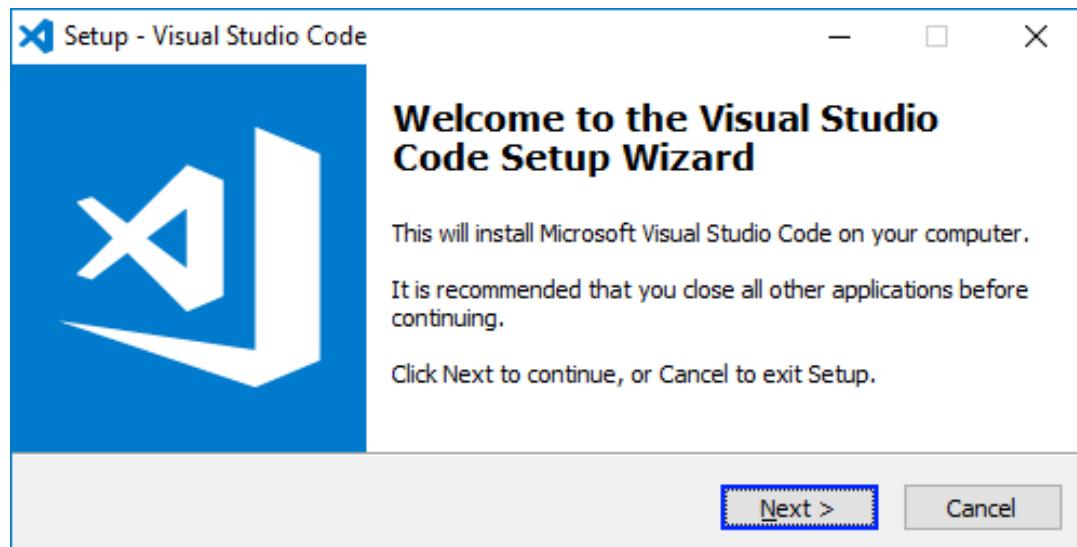
- За програмиране на JavaScript използваме средата **Visual Studio Code**, която се поддържа за операционната система Windows, Linux и Mac OS X.
- Ако програмираме на Java, подходящи са средите **IntelliJ IDEA**, **Eclipse** или **NetBeans**.
- Ако ще пишем на Python, можем да използваме средата **PyCharm**.

## Инсталация на Visual Studio Code

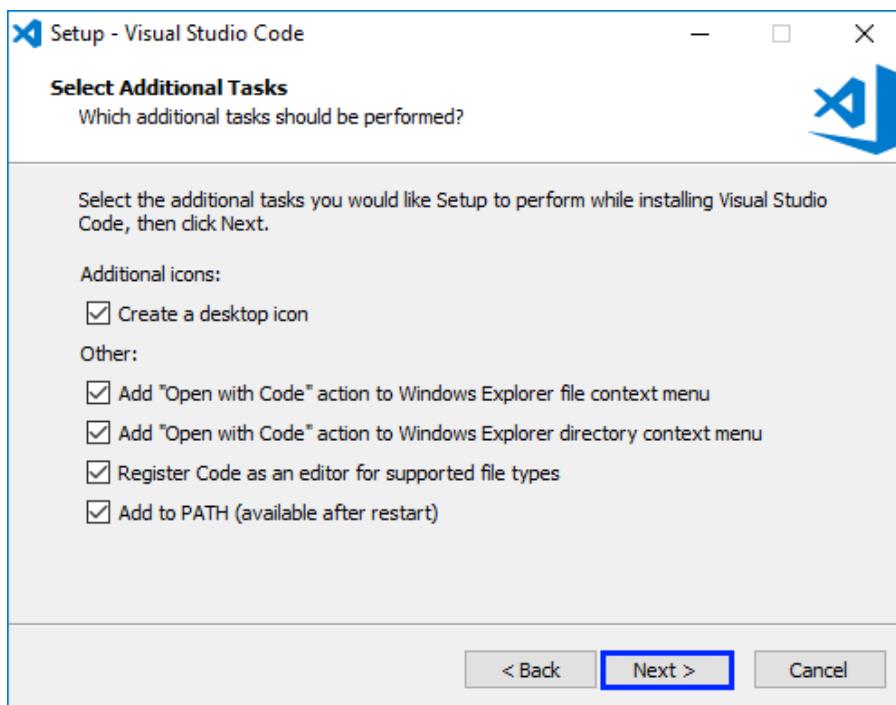
Започваме с инсталацията на интегрираната среда **Microsoft Visual Studio Code** (версия 1.19, актуална към януари 2018 г.).

Visual Studio Code се разпространява бесплатно от Microsoft и може да бъде изтеглено от: <https://code.visualstudio.com>. Инсталацията е типичната за Windows с [Next], [Next] и [Finish]. Не е необходимо да променяме останалите настройки за инсталация.

В следващите редове са описани подробно **стъпките за инсталация на Visual Studio Code** (версия 1.19.1). След като свалим инсталационния файл и го стартираме, се появява следният екран:

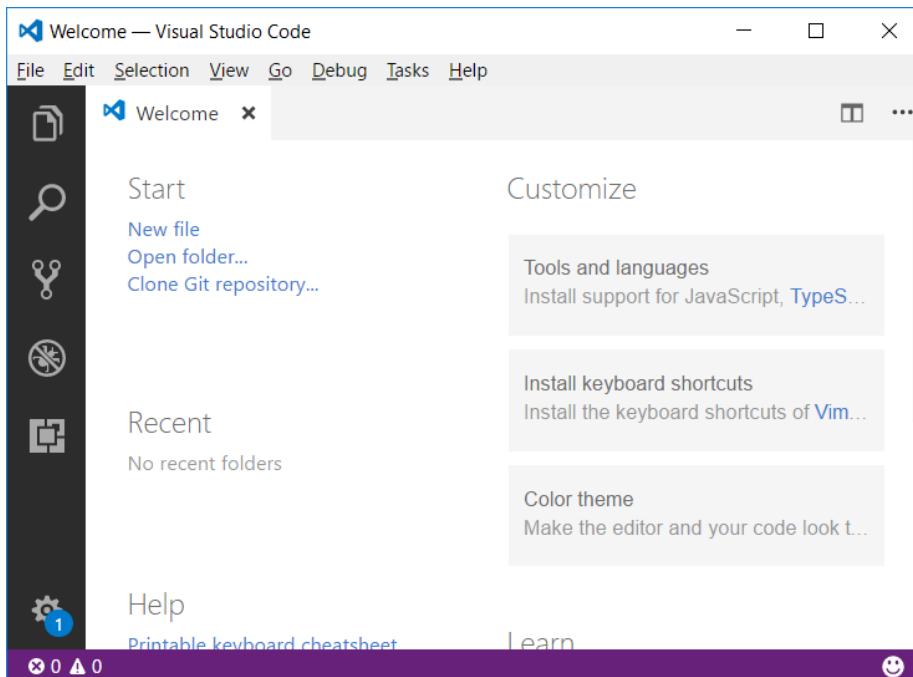


Натискаме бутона **[Next]**, след което ще трява да се съгласим с условията за ползване:



Зарежда се прозорец с инсталационния панел на **Visual Studio Code**, като в даден момент ще бъдем запитани за предпочитанията си към допълнителни настройки, които са си лично индивидуални. Общо взето това е всичко.

Започва инсталацията на **Visual Studio Code** и когато приключи сме почти готови за работи. Накрая, след **старта на VS Code** излиза екран като този по-долу:



Сега е моментът да си направим средата възможно най-приятна за разработка. **Visual Studio Code** е едно от IDE-тата с най-големи възможности за персонализация. Най-често тази персонализация става под формата на **разширения** (extensions). Списък с всички разширения, можете да намерите на официалният сайт <https://marketplace.visualstudio.com>. Имайте предвид, че голяма част от тези разширения са строго специфични при работа с конкретен език за програмиране.

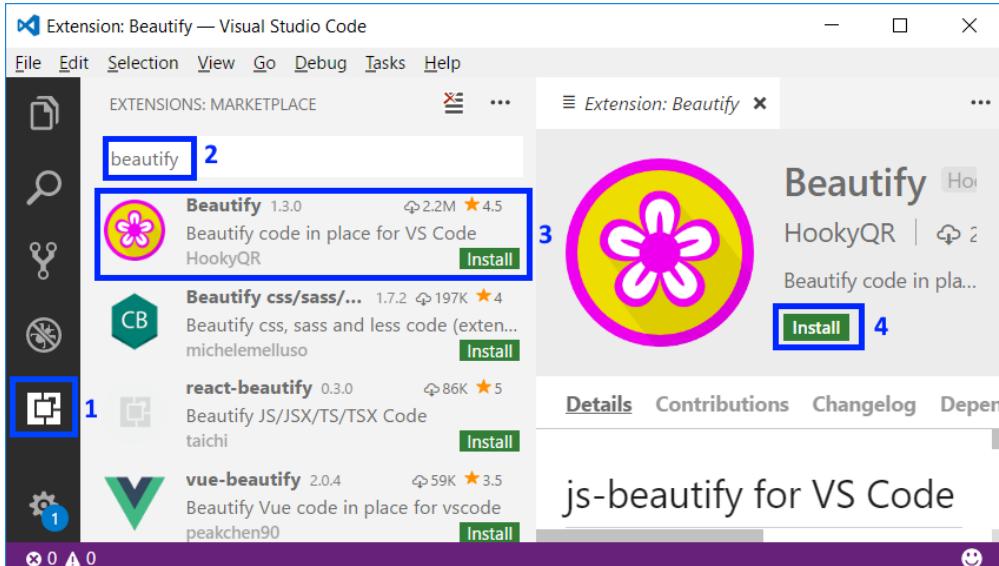
Препоръчваме ви като начало да си инсталирате следните 2 разширения, които драстично ще подобрят работата ви при писане на **JavaScript** код:

- **Beautify** - <https://marketplace.visualstudio.com/items?itemName=HookyQR.beautify> - това е разширение, което ви помага да поддържате кода си чист и подреден.
- **JSHint** - <https://marketplace.visualstudio.com/items?itemName=dbaeumer.jshint> - както казахме, **JavaScript** е **интерпретиран** език и грешките в кода биха се проявили едва след стартирането на изпълнението на програмата. Това разширение търси и съобщава за потенциални нередности по време на писането на кода, преди стартирането на програмата.

Инсталирането става или директно през посочените линкове, или следвайки следните стъпки във **Visual Studio Code**:

1. В най - левият панел, отваряме най-долният таб - Extensions.
2. В полето за търсене изписваме името на разширението, което желаем да инсталираме.
3. От получените резултати избираме този, който ни удовлетворява.
4. Натискаме [**Install**] бутона.

5. Рестартираме **Visual Studio Code**.



Това е всичко. Готови сме за работа с **Visual Studio Code** и **JavaScript**.

## Онлайн среди за разработка

Съществуват и алтернативни среди за разработка онлайн, директно във вашия уеб браузър. Тези среди не са много удобни, но ако нямаете друга възможност, може да стартирате обучението си с тях и да си свалите **Visual Studio Code** по-късно. Такъв примерен сайт е **JSBin** - <https://jsbin.com/?js.console>.

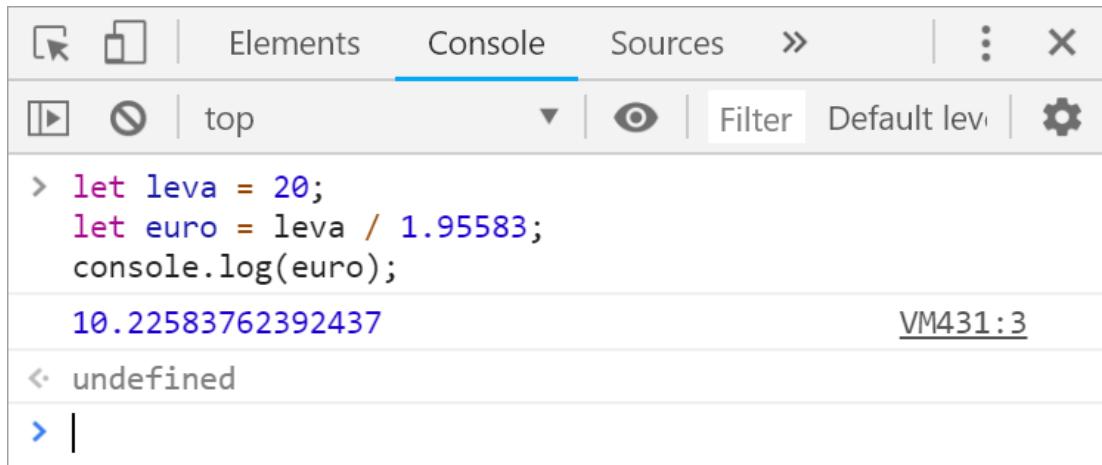


The screenshot shows the JSBin interface. At the top, there's a navigation bar with back, forward, and refresh buttons, followed by a URL field containing <https://jsbin.com/molegihovu/edit?js,console>. Below the URL is a toolbar with tabs for HTML, CSS, JavaScript, Console, and Output. The JavaScript tab is selected, showing the following code:

```
JavaScript ▾  
let leva = 20;  
let euro = leva / 1.95583;  
console.log(euro);
```

To the right of the code editor is a console window with a "Run" button and a "Clear" button. The console displays the output: **10.22583762392437**.

Алтернативно за бързо тестване можем да използваме и директно своят браузър с натискане на **[F12]**, но това генерално е опция, ако искате да изprobвате няколко реда код набързо. Ето пример:

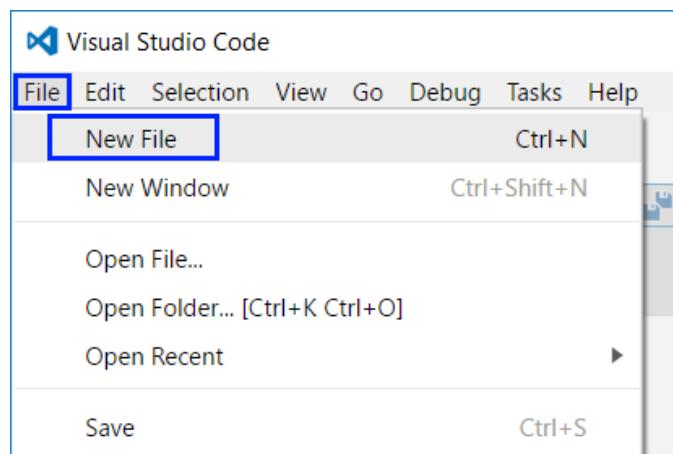


The screenshot shows the Chrome DevTools Console tab. The tab bar includes Elements, Console, Sources, and others. The console output shows the following:

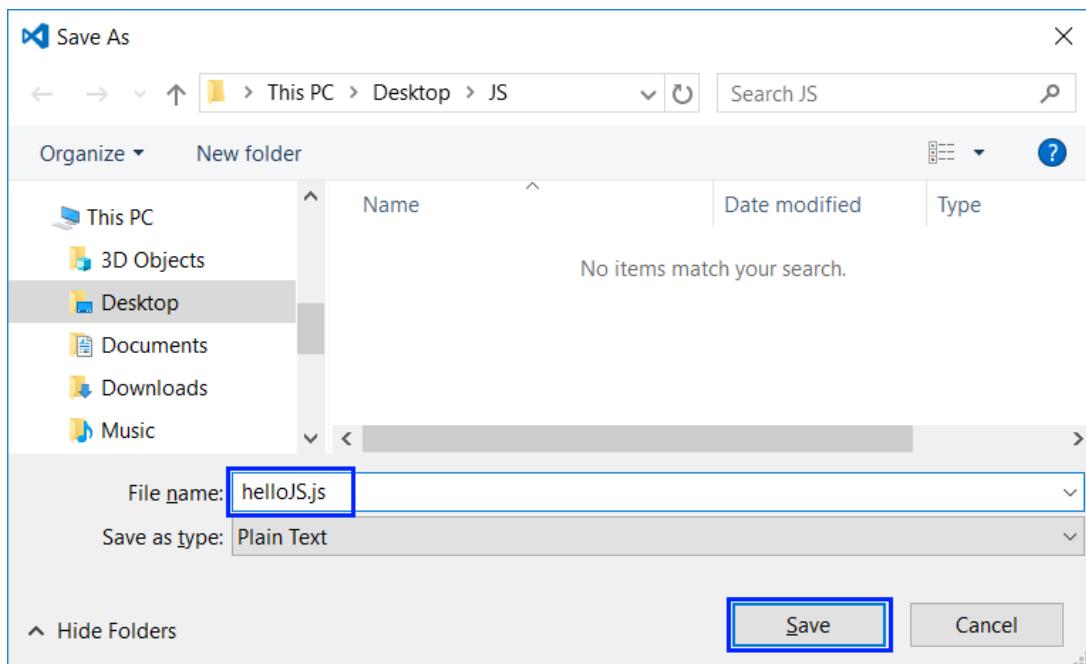
```
> let leva = 20;  
let euro = leva / 1.95583;  
console.log(euro);  
10.22583762392437 VM431:3  
< undefined  
> |
```

## Пример: конзолна програма "Hello JavaScript"

Да се върнем на нашата конзолна програма. Вече имаме **Visual Studio Code** и можем да го стартираме и да пишем JS код в него. След това създаваме нов **JavaScript файл**: **[File] → [New File]**.



След това е важно да запаметим нашият файл от [File] → [Save], като задължително го записваме с разширение **.js**. Също така му задаваме **смислено име**, например **helloJS**:

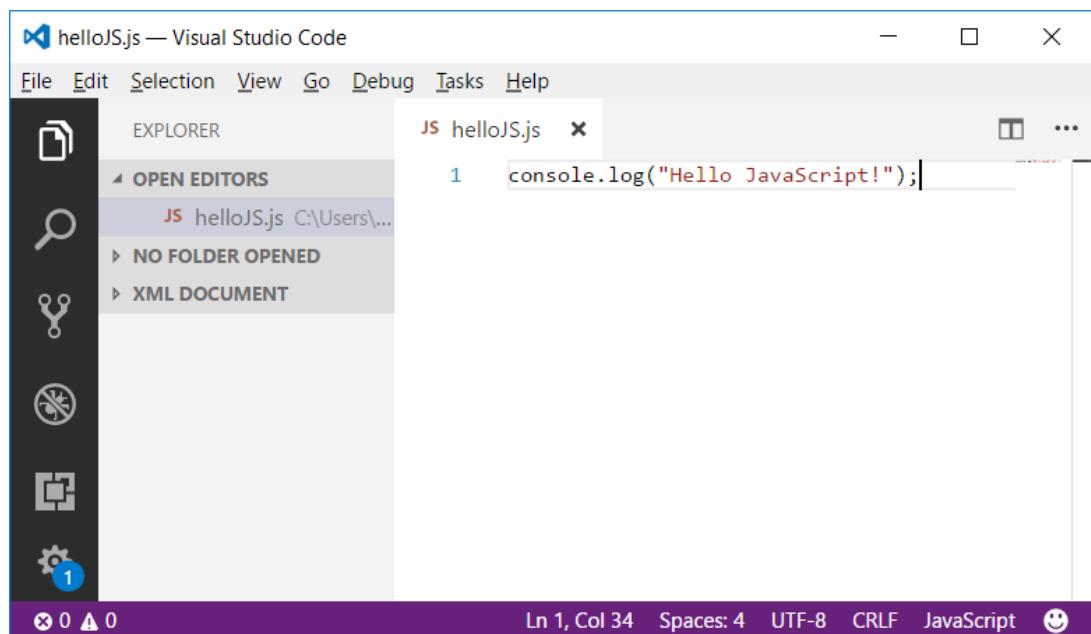


## Писане на програмен код

Писането на **JavaScript** код не изиска никаква допълнителна подготовка от това, което вече направихме - да си създадем файл с разширение **.js**. Затова директно пристъпваме към писането на първият си ред код. Изписваме следната команда:

```
console.log("Hello JavaScript!");
```

Ето как трябва да изглежда нашата програма във **Visual Studio Code**:



Командата **console.log("Hello JavaScript!")** означава да изпълним отпечатване (**log(...)**) върху конзолата (**console**) със съобщение **Hello JavaScript!**, което трябва да оградим с кавички, за да поясним, че това е текст. В края на всяка команда на езика **JavaScript** се слага символът **;** и той указва, че команда свършва на това място (т.е. не продължава на следващия ред). Макар и последното да не е задължително е прието като добра практика, за да се избегнат трудно откривани проблеми.

Тази команда е 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 **log(...)** от класа **console** и му подаваме като параметър текстовия лiteral **"Hello JavaScript!"**.

**Внимание:** използваните преди малко команди **alert(...)** и **prompt(...)** работят сам в уеб браузъра и не са налични при конзолните приложения във VS Code. **Ако се опитате да ги ползвате, ще получите грешка.**

## Стартиране на програмата

За стартиране на програмата натискаме **[F5]** и програмата ще се стартира. Резултатът ще се изпише на конзолата, която за наше удобство ще се отвори директно в долната част на **Visual Studio Code**:

The screenshot shows the Visual Studio Code interface. On the left is the Explorer sidebar with 'OPEN EDITORS' showing 'helloJS.js C:\Users\...'. The main area has a code editor with the following content:

```
1  console.log("Hello JavaScript!");
```

Below the code editor is the Output tab, which displays:

```
Debugging with inspector protocol because Node.js v9.7.1 was detected.
node --inspect=33758 --debug-brk helloJS.js
Debugger listening on ws://127.0.0.1:33758/0da11d11-49fb-42c8-a8aa-340d9d52bb6d
Hello JavaScript!
```

The status bar at the bottom shows 'Ln 1, Col 34 Spaces: 4 UTF-8 CRLF JavaScript'.

Изходът от програмата е следното текстово съобщение:

Hello JavaScript!

Съобщенията "Debugging with inspector protocol ..." и Debugger listening on ... се изписва допълнително на най-горните редове на конзолата от Visual Studio Code след като програмата започне своето изпълнение, като това ни дава допълнителна информация за изпълнението, която за момента можем да игнорираме.

## Тестване на програмата в Judge системата

Тестването на задачите от тази книга е автоматизирано и се осъществява през Интернет, от сайта на Judge системата на СофтУни: <https://judge.softuni.bg>. Оценяването на задачите се извършва в реално време 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/926#0>.

Всеки JavaScript код, който искаме да тестваме в Judge системата трябва да бъде ограден от следните редове допълнителен код:

```
function solve() {
    // we place our code here
}
```

Т.е. ако искаме да тестваме програмата, която току що написахме в системата, тя ще изглежда така:

```
function solve() {
```

```

        console.log("Hello JavaScript");
    }

```

Поставяме целия сурс код на програмата в черното поле и избираме **JavaScript code**, както е показано тук:

**01. Hello JavaScript**

1 function solve() {  
2 console.log("Hello JavaScript");  
3 }

Позволено време: 0.100 sec.      JavaScript code (Node.js)      Изпрати

Позволена памет: 16.00 MB  
Size limit: 16.00 KB  
Checker: Trim

Изпращаме решението за оценяване с бутона **[Изпрати]**. Системата връща резултат след няколко секунди в таблицата с изпратени решения. 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	<b>Детайли</b>
✗ 0 / 100	Памет: 7.22 MB Време: 0.015 s	13:13:30 05.06.2017	<b>Детайли</b>

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 компилация.

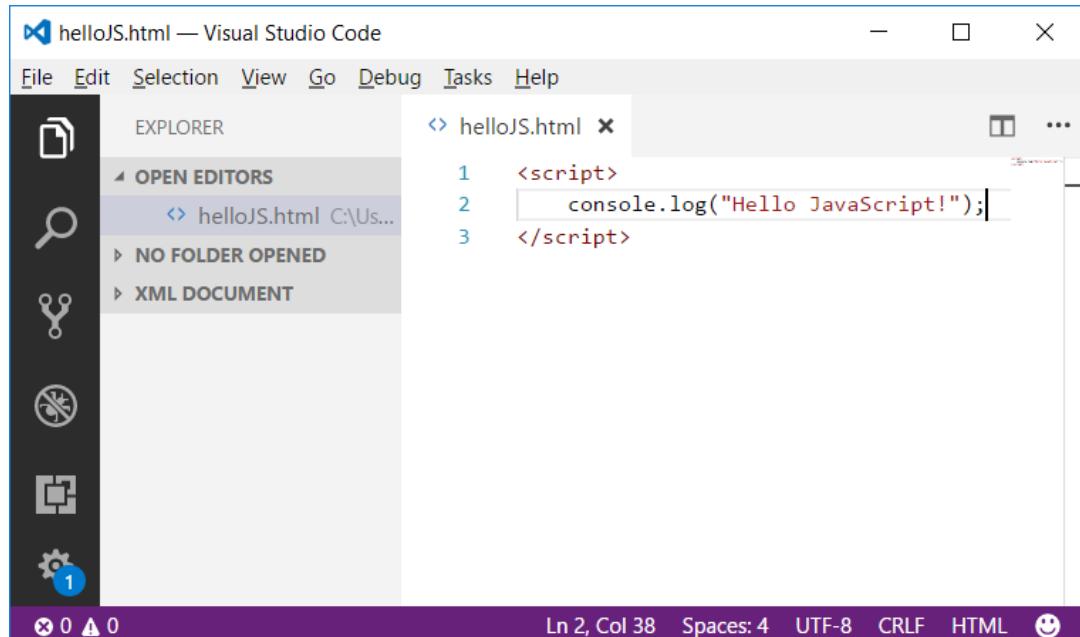
## Как да се регистрирам в SoftUni Judge?

Използваме идентификацията си (username + password) от сайта [softuni.bg](http://softuni.bg). Ако нямате СофтУни регистрация, направете си. Отнема само минутка - стандартна регистрация в Интернет сайт.

## Изпълняване на код в браузър чрез HTML + JS

До тук видяхме как можем да си направим и да изпълним конзолна програма. Нека сега видим как можем да напишем код, който да се изпълнява в нашия браузър. По подобен начин се правят и всички сайтове, които посещавате.

Въщност принципът е много подобен на това, което направихме току-що. Разликата е, че когато си създаваме нов файл, не му даваме разширение **.js**, а **.html**. След това единственото, което трябва да направим е да оградим кодът си със **<script>** отварящият се и **</script>**, затварящ се **HTML таг**. По подобен начин преди малко оградихме кодът си, за да може да бъде тестван в Judge системата. Ето така би изглеждал нашият код във Visual Studio Code сега:



The screenshot shows the Visual Studio Code interface with the following details:

- Title Bar:** helloJS.html — Visual Studio Code
- Menu Bar:** File Edit Selection View Go Debug Tasks Help
- Sidebar:**
  - EXPLORER:** OPEN EDITORS: helloJS.html C:\Us... (highlighted)
  - NO FOLDER OPENED
  - XML DOCUMENT
- Editor Area:**

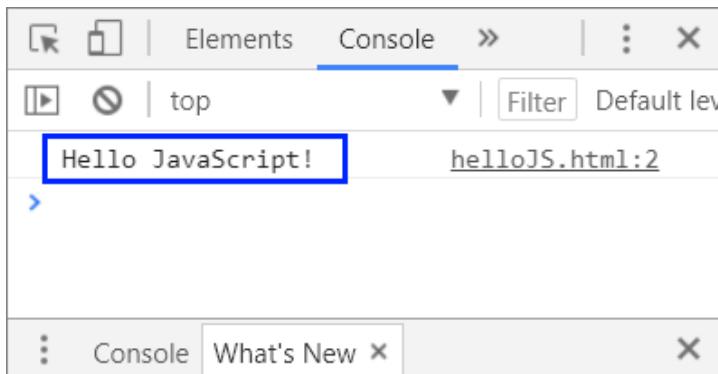
```

1 <script>
2   console.log("Hello JavaScript!");
3 </script>

```
- Bottom Status Bar:** Ln 2, Col 38 Spaces: 4 UTF-8 CRLF HTML 😊

При този подход, сега остава единствено да намерим файла **helloJS.html**, на мястото, където сме го запаметили и да кликнем два пъти върху него. Той ще се зареди в браузъра, но за да видим резултатът от изпълнението му ще трябва да

натиснем [F12], което ще зареди конзолата на браузъра. Ние сме задали команда за печатане на конзолата, затова и трябва да я покажем:



Сега, след като вече **знаете как да изпълнявате програми**, можете да тествате примерните програми по-горе, които показват нотификации на потребителя. Позабавявайте се, пробвайте тези програми. Пробвайте да ги промените и да си поиграете с тях. Заменете командата `console.log("Hello JavaScript");` с команда `console.error("Error occurred");` и стартирайте програмата. Забележете, че програмите с нотификации, могат да бъдат изпълнени само през нашия браузър, а когато опитаме да ги изпълним през конзолата, програмата ще ни даде грешка. Това е така, защото конзолата няма функционалност да нотифицира чрез визуални елементи, какъвто е `alert`.

## Типични грешки в JavaScript програмите

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

```
function solve() {
    Console.Log("Hello JavaScript");
}
```

В горния пример **Console** е изписано грешно и трябва да се поправи на **console**. Каква друга подобна грешка има допусната в програмата?

Липсваща **кавичка** или **липса на отваряща или затваряща скоба** също може да се окажат проблеми. 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:

```
function solve() {
    console.log("Hello JavaScript");
}
```

Тази програма ще даде **грешка след началото на изпълнение** и даже още преди това кодът ще бъде подчертан от разширенията, които следят за това, за да се насочи вниманието на програмиста към грешката, която е допуснал (пропуснатата затваряща кавичка):

```
JS helloJS.js
1 console.log("Hello JavaScript)");
```

[js] Unterminated string literal.  
[jshint] Unclosed string. (W112)

## 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 JavaScript** на базисно ниво и как **да създаваме прости конзолни програми и web програми** с Visual Studio Code. Видяхме как се печата на конзолата с командата **console.log(...)** и как да стартираме програмата си с **[F5]**. Научихме се 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 уеб потребителски интерфейс.

### Задача: конзолна програма “Expression”

Да се напише конзолна **JavaScript** 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).

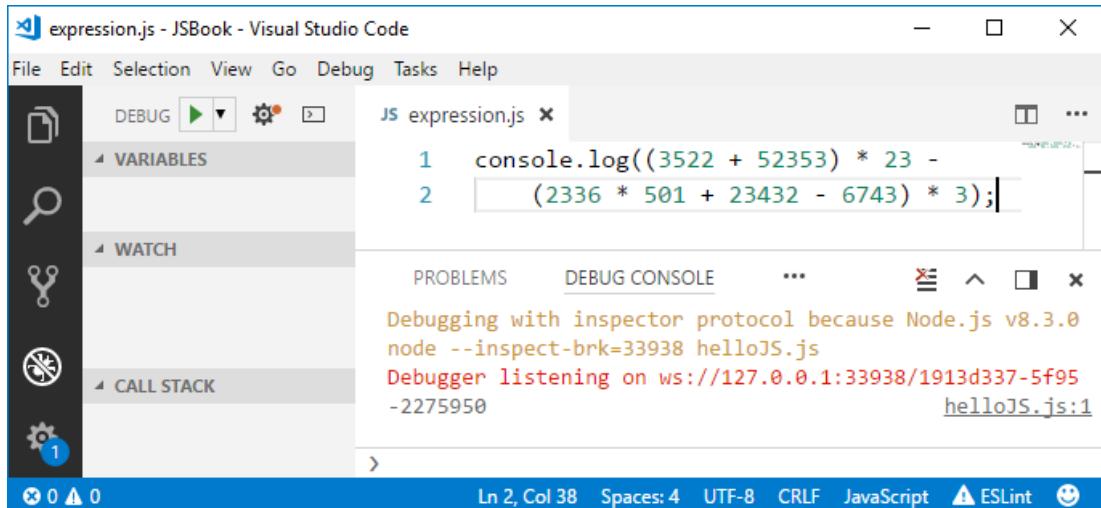
### Насоки и подсказки

Създаваме си **нов JavaScript файл** с име **expression.js**. 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.log(...)`:

```
JS expression.js ✘
1 console.log((3522 + 52353) * 23 -
2           (2336 * 501 + 23432 - 6743) * 3);
```

Стартираме програмата с [F5] и проверяваме дали резултатът е същия като на картинката:



## Тестване в Judge системата

Тествайте решението си: <https://judge.softuni.bg/Contests/Practice/Index/926#1>.

**Забележка:** Не забравяйте да оградите кодът си със `solve()` функцията:

```
function solve() {
    // your code
}
```

## 02. Expression

```
1 function solve() {
2     console.log((3522 + 52353) * 23 - (2336 * 501 + 23432 - 6743) * 3);
```

Резултатът от тестването в judge е нещо такова:

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

JavaScript code (Node.js)

Submissions		
<input type="button" value="&lt;"/> <input type="button" value="&lt;"/> <b>1</b> <input type="button" value="&gt;"/> <input type="button" value="&gt;"/>	<input type="button" value="↻"/>	
Points	Time and memory used	Submission date
✓ 100 / 100	Memory: 11.25 MB Time: 0.234 s	08:25:37 16.03.2018 <input type="button" value="Details"/>
<input type="button" value="&lt;"/> <input type="button" value="&lt;"/> <b>1</b> <input type="button" value="&gt;"/> <input type="button" value="&gt;"/>	<input type="button" value="↻"/>	

## Задача: числата от 1 до 20

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

### Насоки и подсказки

Създаваме нов JavaScript файл с име **nums1To20.js**. В него изписваме 20 команди **console.log(...)**, всяка на отделен ред, за да отпечатаме числата от 1 до 20 едно след друго.

```
nums1To20.js - JSBook - Visual Studio Code
File Edit Selection View Go Debug Tasks Help
DEBUG ▶ [ ] 🔍 ...
VARIABLES
JS nums1To20.js ✘
1 console.log(1);
2 console.log(2);
3 console.log(3);
4 console.log(4);
5 console.log(5);
```

Сега стаптираме програмата и поверьваме дали резултатът е какъвто се очаква да бъде:

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

По-досетливите от вас, сигурно се питат дали няма по-умен начин. Спокойно, има, но за него по-късно.

## Тестване в Judge системата

Тествайте решението си: <https://judge.softuni.bg/Contests/Practice/Index/926#2>.

Получихте ли 100 точки? Ако не сте помислете какво изпускате. А след това помислете дали може да напишем програмата по **по-умен начин**, така че да не повтаряме 20 пъти една и съща команда. Потърсете в Интернет информация за "[for loop JavaScript](#)".

## Задача: триъгълник от 55 звездички

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

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

## Насоки и подсказки

Създаваме **нов JavaScript файл** с име **triangleOf55Stars.js**. В него трябва да напишем код, който печата триъгълника от звездички, например чрез 10 команди, като посочените по-долу:

```
console.log("*");  
console.log("**");  
...  
...
```

## Тестване в Judge системата

Тествайте решението си: <https://judge.softuni.bg/Contests/Practice/Index/926#3>.

Опитайте да **подобрите решението**, така че да няма много повторящи 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). Ако не успеете, няма проблем, ще учим цикли след няколко глави и ще си спомните за тази задача тогава.

## Задача: лице на правоъгълник

Да се напише JavaScript програма, която получава две числа  $a$  и  $b$ , пресмята и отпечатва лицето на правоъгълник със страни  $a$  и  $b$ .

### Примерен вход и изход

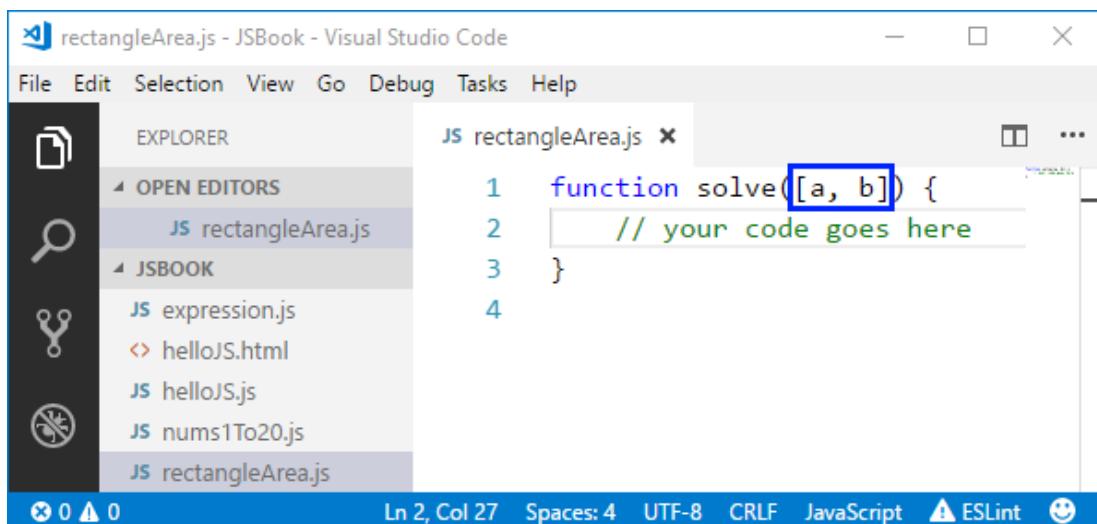
a	b	area
2	7	14

a	b	area
12	5	60

a	b	area
7	8	56

### Насоки и подсказки

Правим нов JavaScript файл. За момента този тип програми ще тестваме само в Judge системата, която има изграден механизъм за подаване на входни данни към програмата. За да получим двете числа, трябва да декларирам това свое желание, като променим ограждащият код (функцията `solve()`), който свикнахме да пишем:



```
function solve([a, b]) {
    // your code goes here
}
```

Забелязахте ли промяната? Между ( и ) поставихме допълнителни квадратни скоби [], между които пък описахме какви данни очакваме да получим - в случая числата **a** и **b**, зададени като масив.

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

### Тестване в Judge системата

Тествайте решението си: <https://judge.softuni.bg/Contests/Practice/Index/926#4>.

\* Задача: квадрат от звездички

Да се напише JavaScript конзолна програма, която получава цяло число N и отпечатва на конзолата квадрат от N звездички, като в примерите по-долу.

### Примерен вход и изход

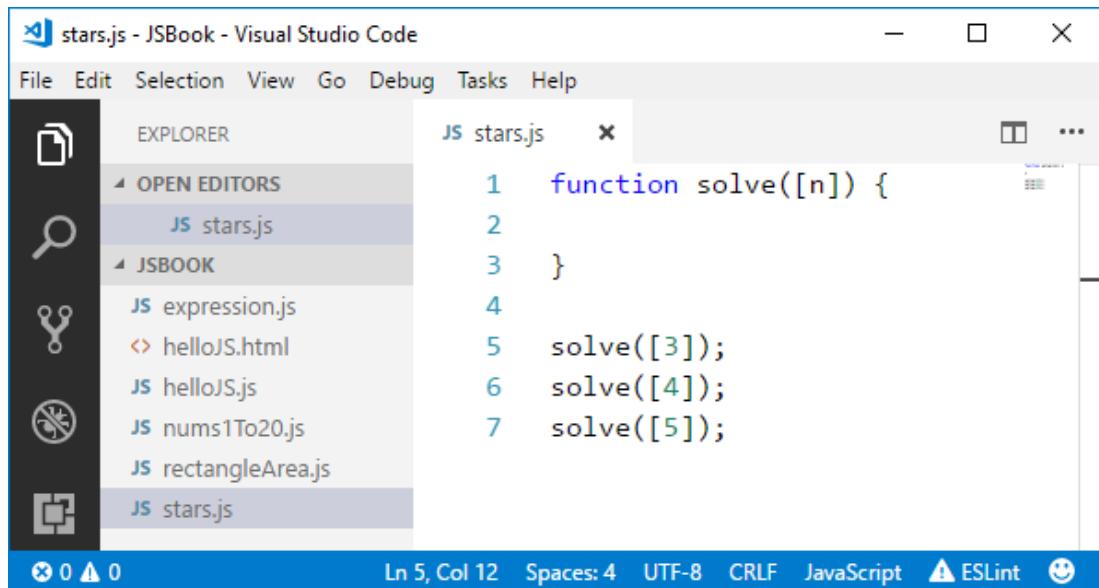
Вход	Изход
3	*** * * ***

Вход	Изход
4	**** * * * * ****

Вход	Изход
5	***** * * * * * * *****

### Насоки и подсказки

Създаваме нова конзолна JavaScript програма:



The screenshot shows the Visual Studio Code interface with the 'stars.js' file open. The code is as follows:

```
function solve([n]) {
}
solve([3]);
solve([4]);
solve([5]);
```

The 'EXPLORER' sidebar shows other files like 'expression.js', 'helloJS.html', 'helloJS.js', 'nums1To20.js', 'rectangleArea.js', and another 'stars.js' file.

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/926#5>.

## Конзолни, графични и уеб приложения

При **конзолните приложения** (Console Applications), както и сами можете да се досетите, **всички операции** за четене на вход и печатане 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 приложението (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 приложението 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. 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 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: уеб приложения

Сега предстои да направим едно просто **уеб 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.

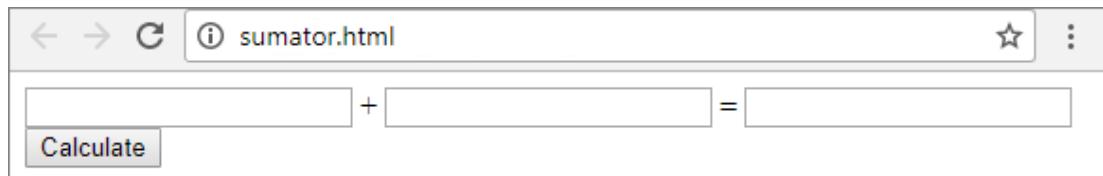
### Задача: Уеб приложение „Суматор за числа“

Да се напише уеб 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. За нашето приложение ще използваме **технологията HTML**, която в комбинация с **езика** за програмиране **JavaScript**, позволява създаване на **web приложения и сайтове**, в среда за разработка **Visual Studio Code**.

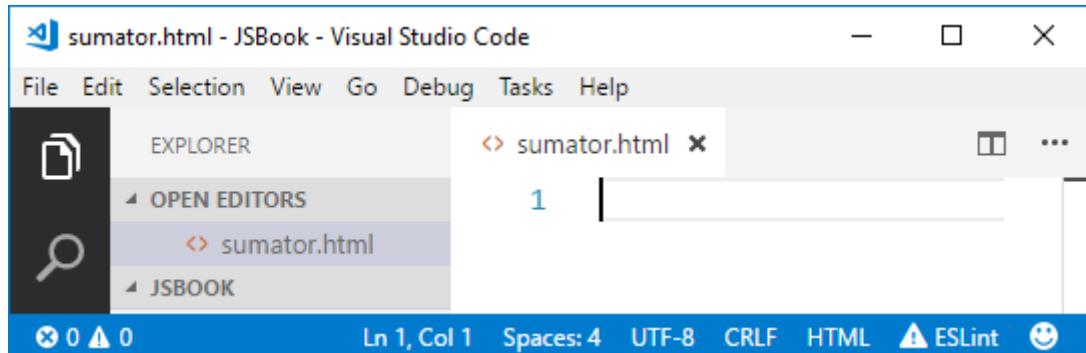
**HTML** е описателен език, чрез който се декларира съдържанието и информацията на даден уеб сайт. Без да задълбаваме просто ще споменем, че **HTML** се базира на използването на комбинации от **тагове**, за да визуализира и предаде **семантичност** на дадено съдържание. В един от предходните примери ние вече създадохме **HTML страница**, като тогава използвахме тагът **<script>**.

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. Уеб приложението ще има сървърна част (back-end), която е написана на езика **JavaScript** и клиентска част (front-end), която е написана на езика **HTML**. Уеб приложението се очаква да изглежда приблизително по следния начин:

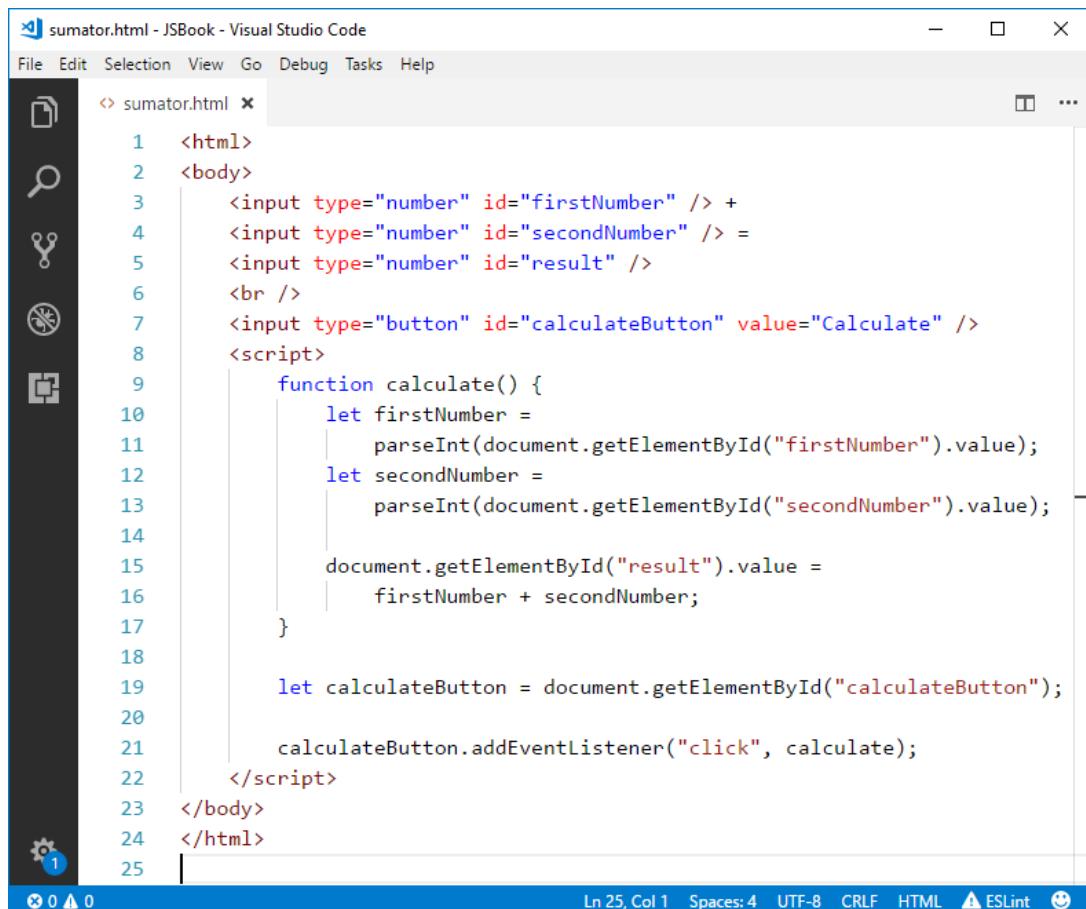


За разлика от конзолните приложения, които четат и пишат данните си във вид на текст на конзолата, уеб приложениета имат **уеб базиран потребителски интерфейс**. Уеб приложениета се **зареждат от някакъв Интернет адрес** (URL) чрез стандартен уеб браузър. Потребителите пишат входните данни в страница, визуализирана от уеб браузъра, данните се обработват на уеб сървър и резултатите се показват отново в страницата в уеб браузъра. Както споменахме за нашето уеб приложение ще използваме **HTML** и **JavaScript**. Други технологии, които ни позволяват създаването на **уеб приложения** са например **технологията ASP.NET MVC**, **технологията PHP** и т.н. Тези технологии улесняват създаването на цялостната структура на приложението - сървърна и клиентска част.

Нека пристъпим към реализирането на нашето приложение. Във **VS Code** създаваме **нов HTML файл**, който кръщаваме **sumator.html**:



Пишем следния код в новосъздадения файл:



The screenshot shows the Visual Studio Code interface with the file 'sumator.html' open. The code is a simple HTML page with a script that adds two numbers input by the user. The code is color-coded for syntax highlighting.

```

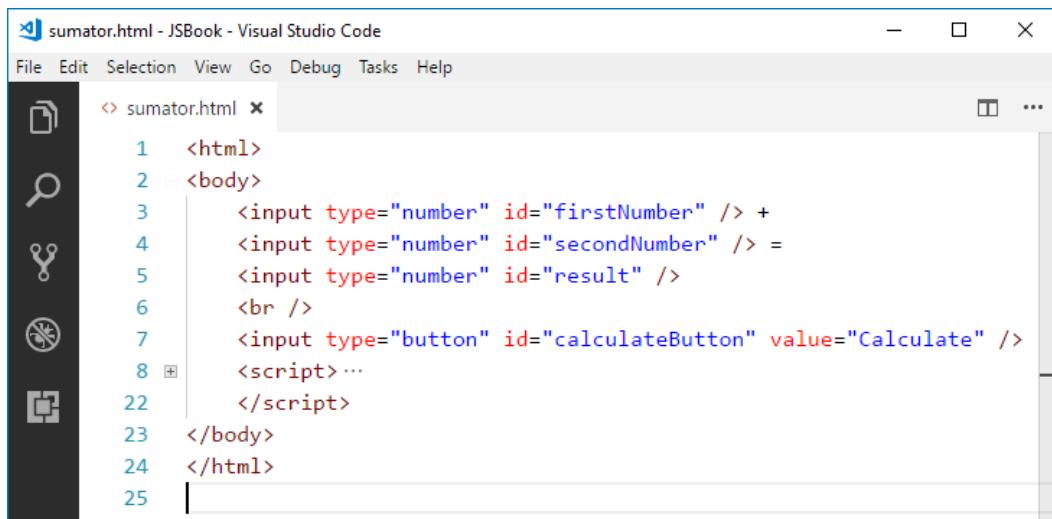
1 <html>
2 <body>
3     <input type="number" id="firstNumber" /> +
4     <input type="number" id="secondNumber" /> =
5     <input type="number" id="result" />
6     <br />
7     <input type="button" id="calculateButton" value="Calculate" />
8     <script>
9         function calculate() {
10             let firstNumber =
11                 parseInt(document.getElementById("firstNumber").value);
12             let secondNumber =
13                 parseInt(document.getElementById("secondNumber").value);
14
15             document.getElementById("result").value =
16                 firstNumber + secondNumber;
17         }
18
19         let calculateButton = document.getElementById("calculateButton");
20
21         calculateButton.addEventListener("click", calculate);
22     </script>
23 </body>
24 </html>
25

```

Този код създава една HTML уеб форма с три текстови полета и един бутона в нея. Указано е, че при натискане на бутона [Calculate] ще се извика действието **calculate**. Нека разгледаме по-подробно кодът, който току що написахме.

На първите 2 реда декларираме, че в момента ще опишем една **HTML страница**, използвайки таговете **<html>** и **<body>**, който декларира започването на тялото на нашата страница - или нейната основна част, която ще бъде визуализирана. На последните 2 реда имаме съответните **затварящи тагове**, които декларират край на областта. Може да забележете, че се различават от **отварящите тагове** по наклонената черта пред името - например **</body>**.

В тялото на нашата страничка, чрез използването на **HTML тагове** описваме това, което искаме да се визуализира, а именно - 3 полета, в които може да се въвеждат числа. Правим това с тага **<input type="number">**. Използвайки този таг ние декларираме, че искаме да имаме визуализация за вход от тип число. Допълнителният **атрибут** - **id** служи за да посочим уникалното име на този таг. Това име е напълно по наш избор. Атрибутът **id** ще ни трява в последствие, за да можем да получим информация за точно този **HTML елемент**.



```

1 <html>
2 <body>
3   <input type="number" id="firstNumber" /> +
4   <input type="number" id="secondNumber" /> =
5   <input type="number" id="result" />
6   <br />
7   <input type="button" id="calculateButton" value="Calculate" />
8   <script>...
9   </script>
10  </body>
11  </html>
12
13
14
15
16
17
18
19
20
21

```

След това на 7-ми ред имаме деклариран още един **input** таг, като този път типът му е **button**. По този начин указваме, че искаме да имаме елемент, върху който може да се натиска и това да доведе до някакви резултати.

Нека сега разгледаме и **JavaScript кодът**, който написахме:

```

9   function calculate() {
10     let firstNumber =
11       parseInt(document.getElementById("firstNumber").value);
12     let secondNumber =
13       parseInt(document.getElementById("secondNumber").value);
14
15     document.getElementById("result").value =
16       firstNumber + secondNumber;
17   }
18
19   let calculateButton = document.getElementById("calculateButton");
20
21   calculateButton.addEventListener("click", calculate);

```

Първо декларираме функция **calculate()**, която прочита информацията от първите две текстови полета (от нашата HTML страница), след което изчислява сумата и я **присвоява** като стойност на третото поле. Функцията продължава от 9-ти до 17-ти ред. Какво представляват **функциите**, как се **декларират** и **извикват**, ще научим малко по-късно в тази книга.

Нека разгледаме по-детайлно **тялото на функцията**. На 10-ти ред **декларираме** променливата **firstNumber**, като ѝ **присвояваме** стойността на текстовото поле с **id=firstNumber**. Функцията **parseInt(...)** ни подсигурява, че въведеният текст ще бъде конвертиран до число. След това, по аналогичен начин получаваме стойността на второто текстово поле (с **id=secondNumber**). Накрая използвайки

същия механизъм, вместо да **вземаме стойността** на третото поле с **id=result**, ние му **присвояваме** стойност, като го поставяме от лявата страна на равенството.

След това, на 19-ти и 20-ти ред достъпваме нашият бутон и му казваме да **слуша за click** събития върху него и когато се получи такова събитие, да се **изпълни** функцията **calculate()**, която **декларирахме** преди малко. С други думи казано, когато кликнем с мишката върху нашият бутон, ще се изпълни функцията **calculate()**.

Приложението е готово. Можем да го стартираме като намерим файлът във файловата система и го отворим. Той ще се зареди в браузъра ни по подразбиране.

Страшно ли изглежда? **Не се плашете!** 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 и уточнения. Или питайте във **форума на СофтУни**: <https://softuni.bg/forum>.

Целта на горният пример (уеб приложение) не е да се научите, а да докоснете по-надълбоко програмирането, да разпалите интереса си към разработката на софтуер и да се вдъхновите да учите здраво. **Имате да учате още много**, но пък е интересно, нали?

# Глава 2.1. Прости пресмятания с числа

В настоящата глава ще се навлезем в следните концепции и програмни техники:

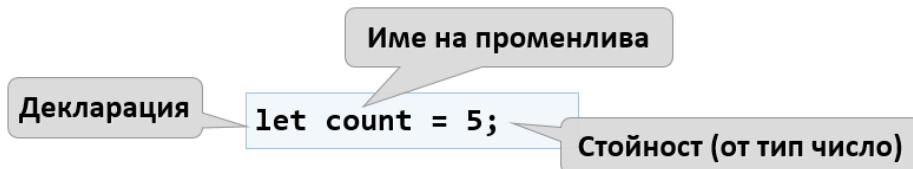
- Как да работим с **типове данни и променливи**, които са ни необходими при обработка на числа и стрингове.
- Как да **изпечатаме** резултат на екрана.
- Как да **четем** потребителски вход.
- Как да извършваме прости **аритметични операции**: събиране, изваждане, умножение, деление, съединяване на стринг.
- Как да **закръгляме** числа.

## Видео

Гледайте видео-урок по тази глава тук: [https://youtu.be/kP\\_1cKnCiA](https://youtu.be/kP_1cKnCiA).

## 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 JavaScript има **име** 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**, **текст** (стринг), **булев** тип, **дата**, **списък** и др. Ето няколко примера за типове данни и стойности за тях:

- **number** - тип число: 1, 42, -5, 3.14, NaN, ...
- **string** - тип текст (стринг): 'Здрави', 'Hi', 'Beer', ...
- **boolean** - булев тип: true, false
- **Date** - дата: Tue Jul 04 2017, ...

В езика **JavaScript** има три ключови думички за деклариране на променлива. Това са **var**, **const** и **let**. Основната разлика между **let** и **var** е в обхвата на

съществуването на променливата. Докато **const** използваме, когато сме сигурни, че това което присвояваме на променливата няма да се променя. Малко по-напред в книгата ще разберем повече подробности за обхвата на променливите, а за сега ще използваме думичката **let**, за да декларираме нова променлива.

## Печатане на резултат на екрана

За да изпечатаме текст, число или друг резултат на екрана, е необходимо да извикаме вградения метод **console.log()**. С него можем да принтираме както стойността на променлива, така и директно текст или число:

```
console.log(42); // печатане на число
console.log('Hello!'); // печатане на текст
let msg = 'Hello, JavaScript!';
console.log(msg); // печатане на стойност на променлива
```

## Четене на потребителски вход – цяло число

За да прочетем потребителски вход под формата на **цяло число**, е необходимо да дефинираме **аргумент** на нашата функция:

```
function sum([arg1, arg2]) {
    let a = parseInt(arg1);
    let b = parseInt(arg2);
    ...
}
```

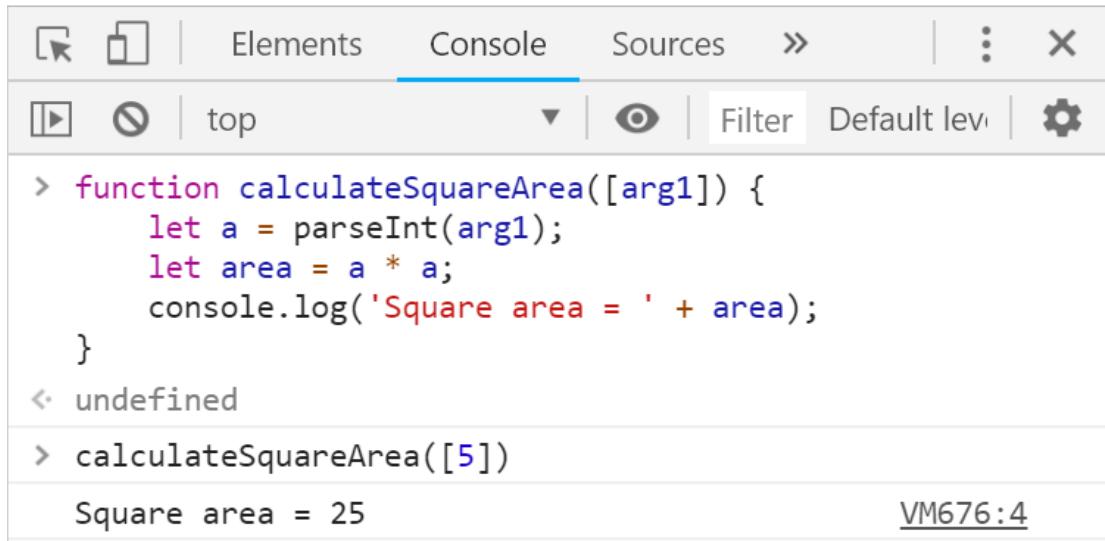
Нека обърнем внимание, че аргументите **arg1** и **arg2** биха могли да бъдат в различен тип данни от този, който желаем. Затова е необходимо да се преобразуват в подходящ за целта такъв. Ако това не се направи, за програмата **всяко едно число** ще бъде просто **текст**, с който **не бихме могли да извършваме** аритметични операции.

## Пример: пресмятане на лице на квадрат със страна a

За пример да вземем следната функция, която чете цяло число, 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:

```
function calculateSquareArea([arg1]) {
    let a = parseInt(arg1);
    let area = a * a;
    console.log('Square area = ' + area);
}
```

Ако извикаме нашата функция с параметър 3, т.е. **calculateSquareArea([3])**, резултатът от кода ще бъде **Square area = 9**. Ето как изглежда нашият код в действие в JavaScript конзолата на уеб браузъра:



The screenshot shows a browser developer console with the "Console" tab selected. The code in the console is:

```
> function calculateSquareArea([arg1]) {
    let a = parseInt(arg1);
    let area = a * a;
    console.log('Square area = ' + area);
}
<- undefined
> calculateSquareArea([5])
Square area = 25
```

On the right side of the console output, the VM identifier is shown as **VM676:4**.

Ако опитаме да въведем невалидно число, например **"hello"**, ще получим съобщение за **грешка** по време на изпълнение (exception). Това е нормално. Покъсно ще разберем как можем да прихващаме такива грешки и да връщаме посмислени за потребителя съобщения.

## Как работи примерът?

На първият ред с **function calculateSquareArea([arg1]) {** дефинираме нашата функция, като и даваме име и задаваме аргументите, от които тя ще се нуждае. В нашия случай имаме един аргумент, който ще представлява страна на квадрата.

На следващият ред с **let a = parseInt(arg1);** взимаме аргумента на функцията **arg1** и го преобразува към цяло число чрез метода **parseInt(arg1);**. Резултатът се записва в променлива с име **a**.

**Забележка:** Ако **arg1** съдържа **дробно число**, то ще бъде преобразувано към **цяло**. Преобразуването на дробно число към цяло става като се **отстранят** цифрите след десетичната запетаята, например: **parseInt(2.3) = 2, parseInt(3.8) = 3**.

Следващата команда **let area = a \* a;** записва в нова променлива, именувана **area**, резултата от умножението на **a** по **a**.

Следващата команда **console.log('Square area = ' + area);** отпечатва посочения текст, като до него долепя изчисленото лице на квадрата, който сме записали в променливата **area**.

Всъщност горната програма може малко да се опрости, ето така:

```
function calculateSquareArea([a]) {
    let area = a * a;
    console.log('Square area = ' + area);
}
```

Горният код ще работи коректно, защото при умножението променливата **a** ще се преобразува към число. Когато входът е само едно единствено число, може да се пропуснат и скобите **[ ]**, ето така:

```
function calculateSquareArea(a) {
    let area = a * a;
    console.log('Square area = ' + area);
}
```

Кодът може да бъде съкратен дори още, ето така:

```
function calculateSquareArea(a) {
    console.log('Square area = ' + a * a);
}
```

## Тестване в Judge системата

Тествайте решението си: <https://judge.softuni.bg/Contests/Practice/Index/927#0>. Пробвайте четирите варианта на решението на задачата.

## Четене на дробно число

За да прочетем потребителски вход под формата на **дробно число**, отново е необходимо да **дефинираме аргумент** на нашата функция. Синтаксисът е подобен както при четене на цяло число, само че тук трябва да използваме функцията **parseFloat(...)**:

```
function sum([arg1, arg2]) {
    let a = parseFloat(arg1);
    let b = parseFloat(arg2);
    ...
}
```

В горния пример параметрите **arg1** и **arg2** се преобразуват в дробни числа в променливите **a** и **b**.

## Пример: прехвърляне от инчове в сантиметри

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

```
function convertInchesToCentimeters([arg1]) {
    let inches = parseFloat(arg1);
    let centimeters = inches * 2.54;
    console.log('Centimeters = ' + centimeters);
}
```

Нека извикаме функцията и да се уверим, че при подаване на стойност в инчове, получаваме коректен резултат в сантиметри:

```
convertInchesToCentimeters([5]); // Centimeters = 12.7
```

## Тестване в Judge системата

Тествайте решението си: <https://judge.softuni.bg/Contests/Practice/Index/927#1>.

## Четене на вход – текст

Както при останалите типове данни, за да прочетем **текст**, е необходимо да **дадем аргумент** на нашата функция, след което да го присвоим на променлива:

```
function print([arg1]) {
    let text = arg1;
    ...
}
```

## Пример: поздрав по име

Да напишем функция, която въвежда името на потребителя и го поздравява с текста "Hello, (име)".

```
function sayHello([arg1]) {
    let name = arg1;
    console.log(`Hello, ${name}!`);
}
```

В този случай, изразът **``${name}``** е заместен от **стойността на променливата `name`**. Ето и резултата, ако извикаме функцията с името "Иван":

```
sayHello(['Иван']); // Hello, Иван!
```

## Тестване в Judge системата

Тествайте решението си: <https://judge.softuni.bg/Contests/Practice/Index/927#2>.

## Съединяване на текст и числа

При печат на текст, числа и други данни, **можем да ги съединим**, като използваме шаблони ``variable = ${variable}``. В програмирането тези шаблони се наричат **placeholders**. Обърнете внимание, че за да бъде разпознат шаблонът, трябва да използваме апостроф ```, вместо обикновени кавички:

```
function printInfo([firstName, lastName, age, town]) {
    console.log(`You are ${firstName} ${lastName}, a ${age}-years old person from ${town}.`);
```

Отново извикваме функцията с тестови параметри и се уверяваме, че работи:

```
printInfo(['Ivan', 'Ivanov', 20, 'Sofia']);
```

Освен променливи, в шаблоните можем да правим и прости изчисления.

Възможно е една и съща променлива да бъде използвана като шаблон повече от веднъж. Ето пример:

```
let a = 1;
console.log(` ${a} + ${a} = ${a + a}`);
```

Резултатът е:

```
1 + 1 = 2
```

## Тестване в Judge системата

Тествайте решението си: <https://judge.softuni.bg/Contests/Practice/Index/927#3>.

## Аритметични операции

Да разгледаме базовите аритметични операции в програмирането.

### Събиране на числа (оператор +)

Можем да събираме числа с оператора `+`:

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

### Изваждане на числа (оператор -)

Изваждането на числа се извършва с оператора `-`:

```
function subtractNumbers([arg1, arg2]) {
```

```

let a = parseInt(a);
let b = parseInt(b);
let result = a - b;
console.log(result);
}

```

Нека проверим резултата от изпълнението на функцията (при числа 10 и 3):

```
subtractNumbers([10, 3]); // 7
```

## Умножение на числа (оператор \*)

Делението на числа се извършва с оператора \*:

```

let a = 5;
let b = 7;
let product = a * b; // 35

```

## Деление на числа (оператор /)

Делението на числа се извършва с оператора /.

**Забележка:** Дробното **деление на 0** не предизвиква грешка, а резултатът е +/-  
безкрайност или специалната стойност **Infinity**.

Ето няколко примера за използване на оператора за делене:

```

console.log(10 / 2.5); // Резултат: 4
console.log(10 / 4); // Резултат: 2.5
console.log(10 / 6); // Резултат: 1.6666666666666667
console.log(a / 0); // Резултат: Infinity
console.log(-a / 0); // Резултат: -Infinity

console.log(0 / 0); // Резултат: NaN (Not a Number), т.е.
// резултатът от операцията не е валидна
// числена стойност

```

## 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 +:

```

let firstName = "Maria";
let lastName = "Ivanova";
let age = 19;

```

```
let str = firstName + " " + lastName + " @ " + age;
console.log(str); // Maria Ivanova @ 19
```

Ето още един пример:

```
let a = 1.5;
let b = 2.5;
let sum = "The sum is: " + a + b;
console.log(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:

```
let a = 1.5;
let b = 2.5;
let sum = "The sum is: " + (a + b);
console.log(sum); // The sum is: 4
```

## Числени изрази

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

```
let 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 функцията, която приема дълчините на двете основи на трапец и неговата височина (по едно дробно число на ред) и пресмята **лицето на трапеца** по стандартната математическа формула:

```
function printTrapezoidArea([arg1, arg2, arg3]) {
    let b1 = parseFloat(arg1);
    let b2 = parseFloat(arg2);
    let h = parseFloat(arg3);
    let area = (b1 + b2) * h / 2;
    console.log("Trapezoid area = " + area);
}
```

Тъй като искаме функцията ни да работи, както с цели, така и с дробни числа, използваме **`parseFloat()`**. Ако стартираме функцията и въведем за страните съответно **3, 4 и 5**, ще получим следния резултат:

```
printTrapezoidArea([3, 4, 5]); // Trapezoid area = 17.5
```

## Тестване в Judge системата

Тествайте решението си: <https://judge.softuni.bg/Contests/Practice/Index/927#4>.

## Закръгляне на числа

Понякога, когато работим с дробни числа, се налага да приведем числата към еднотипен формат. Това привеждане се нарича **закръгляне**. Езикът **JavaScript** ни предоставя няколко метода за закръгляне на числа:

- **`Math.ceil(...)`** - закръгляне нагоре, до следващо (по-голямо) цяло число:

```
let up = Math.ceil(45.15); // up = 46
```

- **`Math.floor(...)`** - закръгляне надолу, до предишно (по-малко) цяло число:

```
let down = Math.floor(45.67); // down = 45
```

- **`Math.trunc(...)`** - отрязване на знаците след десетичната запетая:

```
let trunc = Math.trunc(45.67); // trunc = 45
```

- **`Math.round(...)`** - закръглянето се извършва по **основното правило за закръгляване** - ако десетичната част е по-малка от 5, закръглението е надолу и обратно, ако е по-голяма от 5 - нагоре:

```
Math.round(5.439); // 5
```

```
Math.round(5.539); // 6
```

- **`.toFixed([брой символи след десетичната запетая])`** - закръгляне до най-близко число:

```
(123.456).toFixed(2); // 123.46
```

```
(123).toFixed(2); // 123.00
```

```
(123.456).toFixed(0); // 123
```

```
(123.512).toFixed(0); // 124
```

## Пример: периметър и лице на кръг

Нека напишем функция, която приема радиуса  $r$  на кръг и изчислява лицето и периметъра на кръга / окръжността.

Формули:

- Лице =  $\pi * r * r$
- Периметър =  $2 * \pi * r$

- $\pi \approx 3.14159265358979323846\ldots$

```
function calculateCircleAreaAndPerimeter([arg1]) {
    let r = parseInt(arg1);
    console.log("Area = " + Math.PI * r * r);
    // Math.PI - вградена в JavaScript константа за стойността
    // на числото π
    console.log("Perimeter = " + 2 * Math.PI * r);
}
```

Нека извикаме функцията с радиус **r = 10**:

```
calculateCircleAreaAndPerimeter([10])
```

Резултатът е следният:

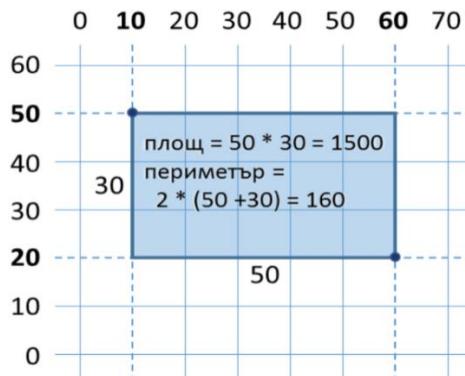
Area = 314.1592653589793
Perimeter = 62.83185307179586

## Тестване в Judge системата

Тествайте решението си: <https://judge.softuni.bg/Contests/Practice/Index/927#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-голямата **x** координата извадим по-малката **x** координата, 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:

```
function calculateRectangleArea([arg1, arg2, arg3, arg4]) {
    let x1 = parseFloat(arg1);
    let y1 = parseFloat(arg2);
```

```

let x2 = parseFloat(arg3);
let y2 = parseFloat(arg4);

// Изчисляване страните на правоъгълника:
let width = Math.max(x1, x2) - Math.min(x1, x2);
let height = Math.max(y1, y2) - Math.min(y1, y2);

console.log(width * height);
console.log(2 * (width + height));
}

```

Използваме метода **Math.max(x1, x2)**, за да намерим по-голямата измежду стойностите **x1** и **x2** и аналогично **Math.min(y1, y2)** за намиране на по-малката от двете стойности.

Нека извикаме функцията с тестови стойности от координатната система:

```

calculateRectangleArea([60, 20, 10, 50]); // 1500
                                         // 160

```

## Тестване в Judge системата

Тествайте решението си: <https://judge.softuni.bg/Contests/Practice/Index/927#6>.

## Какво научихме от тази глава?

Да резюмираме какво научихме от тази глава на книгата:

- Четене на потребителски вход:
  - **function sum([number1, number2])**
- Преобразуване към число:
  - **let num = parseInt(arg1)**
  - **let num = parseFloat(arg1).**
- Извършване на пресмятания с числа и използване на съответните аритметични оператори [+,-,\*,/,(,)]:
  - **let sum = 5 + 3.**
- Извеждане на текст по шаблон:
  - **console.log(`3 + 5 = \${3 + 5}`).**
- Различните типове закръгляния на числа: **Math.ceil()**, **Math.trunc()**, **Math.floor()** и **.toFixed()**

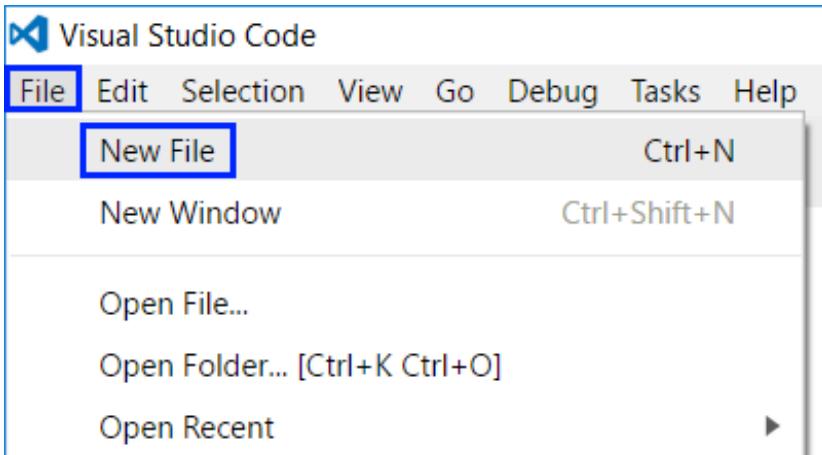
## Упражнения: прости пресмятания

Нека затвърдим наученото в тази глава с няколко задачи.

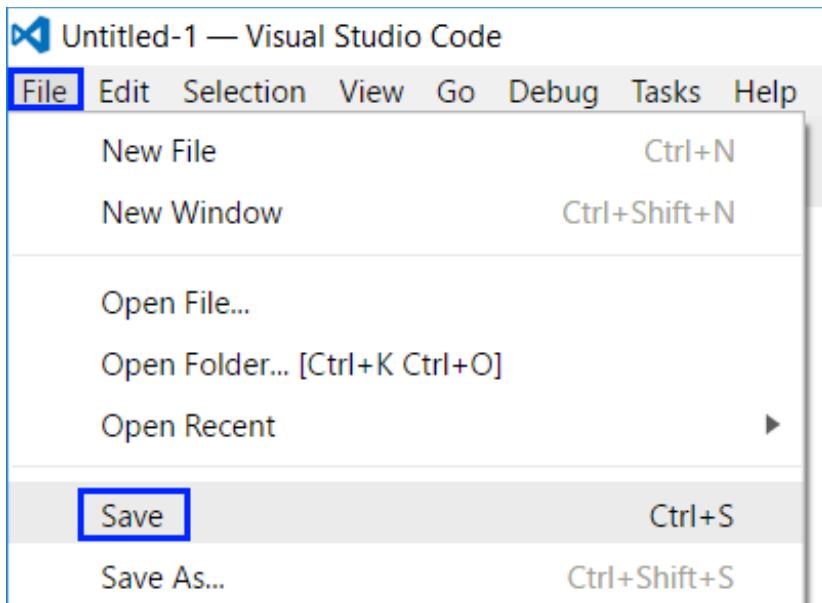
## Празен JS файл за решението на задачата ни във VS Code

Започваме като създадем празен **JS файл** във Visual Studio Code. В настоящото практическо занимание ще създадем нова папка и ще добавяме нов **JS файл** за всяка задача, за да организираме решенията на задачите от упражненията:

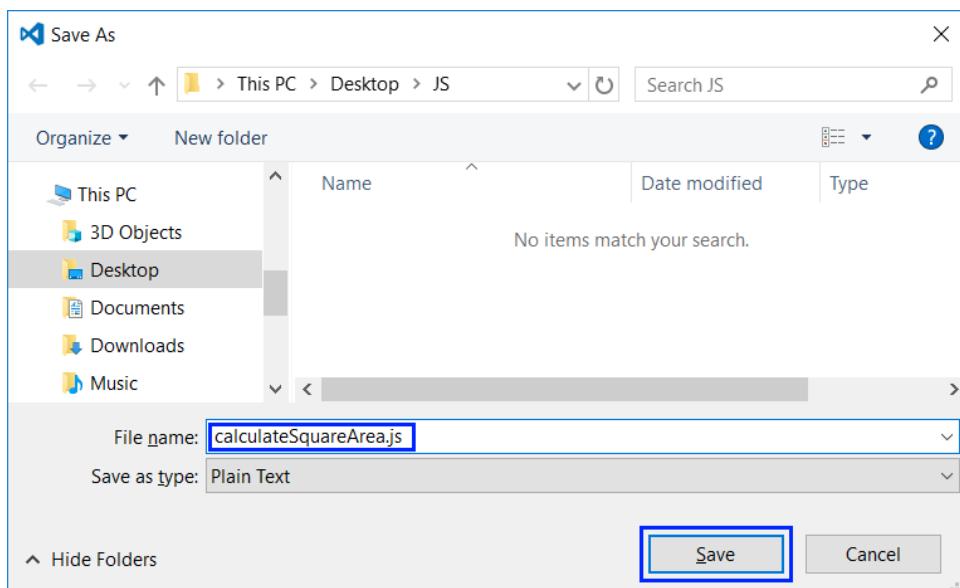
Стартираме Visual Studio Code и създаваме **нов файл**: [File] -> [New File]:



Запаметяваме файла от [File] -> [Save] или чрез клавишната комбинация [Ctrl + S]:



Даваме **значещо име** и разширение **.js** на нашия файл, след което натискаме бутона **[Save]**:



## Задача: пресмятане на лице на квадрат

Първата задача от тази тема е следната: да се напише функция, която **получава** цяло число **a** и пресмята лицето на квадрат със страна **a**. Задачата е тривиално лесна: приемате число като аргумент на функцията, умножавате го само по себе си и печатате получения резултат на конзолата.

### Насоки и подсказки

Вече имаме правилно именуван празен файл. Остава да напишем кода за решаване на задачата. За целта пишем следния код:

```
function calculateSquareArea([arg1]) {
    let a = parseInt(arg1);
    let area = a * a;
    console.log(area);
}
```

Кодът дефинира функция **calculateSquareArea()**, която приема един аргумент **arg1**. Тъй като се очаква аргументът да е цяло число, преобразуваме аргумента с метода **parseInt()** и след това изчисляваме лицето: **area = a \* a**. Накрая печатаме стойността на променливата **area**. За да **тестваме**, е нужно в същия файл да **извикаме функцията** с произволен параметър и след това да стартираме програмата като натиснем **[Ctrl + F5]**:

The screenshot shows a terminal window with tabs for 'PROBLEMS', 'OUTPUT', 'DEBUG CONSOLE', and 'TERMINAL'. The 'TERMINAL' tab is active. In the terminal, the command `calculateSquareArea([5]);` is typed and executed. The output is 'Square area = 25'.

## Тестване в Judge системата

Тествайте решението си: <https://judge.softuni.bg/Contests/Practice/Index/927#0>.

Трябва да получите 100 точки (напълно коректно решение):

### 01. Square Area

```
1 function calculateSquareArea([arg1]) {
2     let a = parseInt(arg1);
3     let area = a * a;
4     console.log('Square area = ' + area);
5 }
```

Allowed working time: 0.100 sec.

Allowed memory: 16.00 MB

Size limit: 16.00 KB

Checker: Numbers Checker 

JavaScript code (Node.js)

**Submit**

Submissions		
Points	Time and memory used	Submission date
    100 / 100	Memory: 11.16 MB Time: 0.025 s	15:16:44 04.02.2018 

## Задача: от инчове към сантиметри

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

### Насоки и подсказки

Първо създаваме нов файл в папката с другите решения - във Visual Studio Code избираме [File] -> [New file]. Запаметяваме файла под някакво добро име, например **convertInchesToCentimeters.js** и натискаме бутона [Save]. Следва да напишем кода на програмата:

```
function convertInchesToCentimeters([arg1]) {
    let inches = parseFloat(arg1);
    let centimeters = inches * 2.54;
    console.log('Centimeters = ' + centimeters);
}
```

Извикваме функцията с параметър 2 и стартираме програмата с [Ctrl + F5]:

```
7 convertInchesToCentimeters([2]);
```

PROBLEMS      OUTPUT      DEBUG CONSOLE      TERMINAL

Centimeters = 5.08

Да тестваме с дробни числа, например с 4.5:

```
7 convertInchesToCentimeters([4.5]);
```

PROBLEMS      OUTPUT      DEBUG CONSOLE      TERMINAL

Centimeters = 11.43

## Тестване в Judge системата

Тествайте решението си: <https://judge.softuni.bg/Contests/Practice/Index/927#1>.

Решението би трябвало да бъде прието като напълно коректно:

Submissions		
1		
Points ✓✓✓✓ 100 / 100	Time and memory used Memory: 11.46 MB Time: 0.093 s	Submission date 14:47:08 04.02.2018

## Задача: поздрав по име

Да се напише функция, която приема като аргумент име на човек и отпечатва **Hello, <name>!**, където **<name>** е въведеното преди това име.

### Насоки и подсказки

Отново създаваме **нов файл** в папката с другите решения и го запазваме под името **sayHello.js**. Следва да напишем кода на програмата. Ако се затруднявате, може да ползвате примерния код по-долу:

```
function sayHello([arg1]) {
    let name = arg1;
    console.log(`Hello, ${name}!`);
```

Извикваме функцията с примерен параметър и **стартираме програмата** с [Ctrl+F5], за да тестваме дали работи коректно:

```
6 sayHello(['Ivan']);
```

PROBLEMS    OUTPUT    DEBUG CONSOLE    TERMINAL

Hello, Ivan!

## Тестване в Judge системата

Тествайте решението си: <https://judge.softuni.bg/Contests/Practice/Index/927#2>.

## Задача: съединяване на текст и числа

Напишете функция, която получава като аргумент 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

По същия начин създаваме нов файл и го именуваме **printInfo.js**. Кодът, който отпечатва описаното в условието на задачата съобщение, е целенасочено размазан и трябва да се допише от читателя:

```
function printInfo([firstName, lastName, age, town]) {  
    console.log(`You are ${firstName} ${lastName}, a ${age}-years old person from ${town}.`);  
}
```

Следва да се тества решението локално, като се извика функцията с примерни стойности и се стартира програмата с **[Ctrl+F5]**.

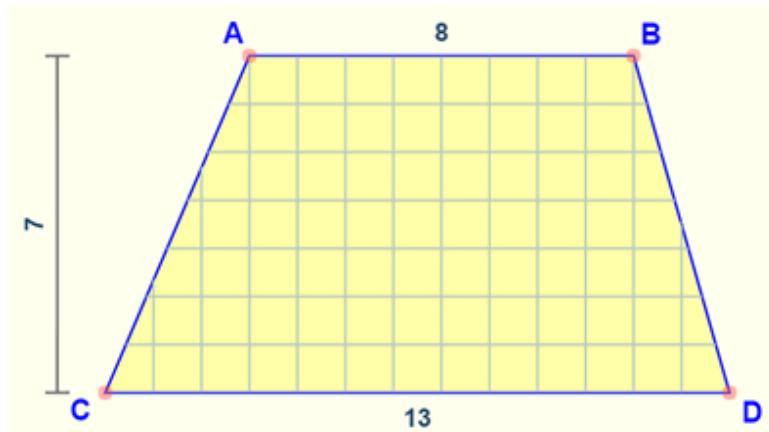
## Тестване в Judge системата

Тествайте решението си: <https://judge.softuni.bg/Contests/Practice/Index/927#3>.

## Задача: лице на трапец

Напишете функция, която получава аргумент три числа **b1**, **b2** и **h** и пресмята лицето на трапец с основи **b1** и **b2** и височина **h**. Формулата за лице на трапец е  $(b1 + b2) * h / 2$ .

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 във Visual Studio Code файл с име **calculateTrapezoidArea.js** и да напишем кода, който чете входните данни от аргументите на функция, пресмята лицето на трапеца и го отпечатва. Кодът на картинката е нарочно размазан, за да помисли читателят върху него и да го допише сам:

```
function calculateTrapezoidArea([arg1, arg2, arg3]) {
    let b1 = parseFloat(arg1);
    // Възможно е тук да има проблеми
    // Възможно е тук да има проблеми
    // ако е това е така + то
    console.log("Trapezoid area = " + area);
}
```

Тествайте решението локално с извикване на функцията и стартиране с [Ctrl+F5].

### Тестване в Judge системата

Тествайте решението си: <https://judge.softuni.bg/Contests/Practice/Index/927#4>.

### Задача: периметър и лице на кръг

Напишете функция, която получава аргумент **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	4.5	Area = 63.6172512351933 Perimeter = 28.2743338823081

## Насоки и подсказки

За изчисленията можете да използвате следните формули:

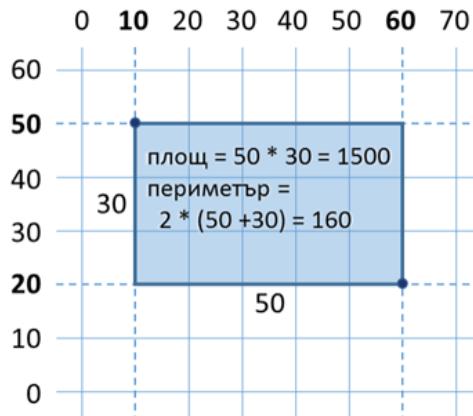
- **Area = Math.PI \* r \* r.**
- **Perimeter = 2 \* Math.PI \* r.**

## Тестване в Judge системата

Тествайте решението си: <https://judge.softuni.bg/Contests/Practice/Index/927#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 пресметнат площта и периметъра му. Входът се приема като аргумент на функция. 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	Вход	Изход	Вход	Изход
60		30		600.25	350449.6875
20	1500	40	2000	500.75	2402
10	160	70	180	100.50	
50		-10		-200.5	

## Тестване в Judge системата

Тествайте решението си: <https://judge.softuni.bg/Contests/Practice/Index/927#6>.

## Задача: лице на триъгълник

Напишете функция, която приема като аргумент страна и височина на триъгълник и пресмята неговото лице. Използвайте формулата за лице на триъгълник:  $area =$

$a * h / 2$ . Закръглете резултата до 2 цифри след десетичния знак, използвайки `area.toFixed(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/927#7>.

### Задача: конвертор - от градуси °C към градуси °F

Напишете функция, която чете градуси по скалата на Целзий ( $^{\circ}\text{C}$ ) и ги преобразува до градуси по скалата на Фаренхайт ( $^{\circ}\text{F}$ ). 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	Вход	Изход	Вход	Изход	Вход	Изход
25	77	0	32	-5.5	22.1	32.3	90.14

### Тестване в Judge системата

Тествайте решението си: <https://judge.softuni.bg/Contests/Practice/Index/927#8>.

### Задача: конвертор - от радиани в градуси

Напишете функция, която чете ъгъл в [радиани](#) (`rad`) и го преобразува в [градуси](#) (`deg`). Потърсете в Интернет подходяща формула. Числото  $\pi$  в JavaScript 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/927#9>.

### Задача: конвертор - USD към BGN

Напишете функция 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	Вход	Изход	Вход	Изход
20	35.91 BGN	100	179.55 BGN	12.5	22.44 BGN

## Тестване в Judge системата

Тествайте решението си: <https://judge.softuni.bg/Contests/Practice/Index/927#10>.

### Задача: \* междувалутен конвертор

Напишете функция за конвертиране на парична сума от една валута в друга. Трябва да се поддържат следните валути: BGN, USD, EUR, GBP. Използвайте следните фиксираны 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	Вход	Изход
20 USD BGN	35.91 BGN	12.35 EUR GBP	9.53 GBP
100 BGN EUR	51.13 EUR	150.35 USD EUR	138.02 EUR

## Тестване в Judge системата

Тествайте решението си: <https://judge.softuni.bg/Contests/Practice/Index/927#11>.

## Задача: \*\* пресмятане с дати - 1000 дни на Земята

Напишете функция, която чете **ръждена дата** във формат **dd-MM-yyyy** и пресмята датата, на която се навършват **1000 дни** от тази ръждена дата и я отпечатва в същия формат.

### Примерен вход и изход

Вход	Изход	Вход	Изход	Вход	Изход
25-02-1995	20-11-1997	14-06-1980	10-03-1983	30-12-2002	24-09-2005
07-11-2003	02-08-2006	01-01-2012	26-09-2014		

### Насоки и подсказки

- Потърсете информация за типа **Date** в JavaScript и по-конкретно разгледайте методите  **setDate(...)**,  **getDate()**,  **getMonth()** и  **getYear()**. 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/927#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 **уеб приложение** с графичен потребителски интерфейс (GUI) с пресмятания с дробни числа.

### Уеб приложение: \*\*\* конвертор от BGN към EUR!

Създайте уеб приложение, което пресмята стойността в **евро** (EUR) на парична сума, зададена в **лева** (BGN).

BGN to EUR Converter

BGN: 54.50

EUR: 27.87

Convert!

При промяна на стойността в лева, равностойността в евро трябва да се преизчислява автоматично. Използвайте курс лева / евро: **1.95583**.

По подобен начин както в първата глава "[Първи стъпки в програмирането](#)", за нашето приложение ще използваме езиците **JavaScript, HTML и CSS**.

1. Първата стъпка е да си **създадем папка** в която ще съхраняваме всички файлове които са необходими за нашето приложение.
2. След това в папката трябва да създадем HTML файл: **index.html**

```
<!DOCTYPE html>
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
    <meta charset="utf-8">
    <title>BGN to EUR Converter</title>
</head>
<body>
    <form class="content-form">
        <h2 class="title">BGN to EUR Converter</h2>
        <section class="items">
            <label for="bgn" class="currency">
                <span class="item-currency">BGN: </span>
                <input class="currency-value" type="number"
                    id="bgn" value="0" />
            </label>
            <label for="euro" class="currency">
                <span class="item-currency">EUR: </span>
                <input class="currency-value" type="text"
                    id="euro" readonly />
            </label>
            <input class="primary-btn" type="button"
                value="Convert!" />
        </section>
    </form>
</body>
</html>
```

Обърнете внимание, че всяка HTML страница трябва да има **определената структура**. Например винаги основният код който пишем е в тага **<body>**, и винаги заглавието на страницата е в тага **<title>**.

3. Вече имаме структурата на страницата, остава да добавим и **JavaScript** файл със самата логика. Създаваме нов файл и го именуваме **converter.js**.

```
function eurConverter() {
    let bgn = document.getElementById("bgn").value;
    let eur = (bgn / 1.95583).toFixed(2);
    document.getElementById("euro").value = eur;
```

```
}
```

4. След като имаме логиката на приложението, трява да намерим начин да кажем къде да се използва. За целта трява да направим 2 промени в съществуващия `index.html` файл:

Първо добавяме следния ред точно под `title` тага, чрез който се осъществява връзката между файловете `index.html` и `converter.js`:

```
<script src="converter.js" type="text/javascript"></script>
```

И второ, намираме и заместваме `input` полето с тип `button` със следния код. По този начин задаваме **при клик** на бутона **[Convert!]** да се извика функцията `eurConverter()`:

```
<input class="primary-btn" type="button"
onclick="eurConverter()" value="Convert!" />
```

Ако стартираме файла `index.html` от папката, в момента би трявало да имаме работещо приложение, което да конвертира от BGN към EUR:



Нека го направим по-красиво.

5. Създаваме нов файл с разширение `.css` и име `index.css` служи за стилизиране на елементите в HTML. Отваряме файла `index.html` и добавяме следния ред в тага `<head>`:

```
<link rel="stylesheet" href="index.css" type="text/css" />
```

Във файла `index.css` слагаме следния код (дефинираме стилове за отделните елементи от HTML формата):

```
body {
    font-family: 'Lato', sans-serif;
    color: #FFFFFF;
}

.content-form {
    width: 50%;
    margin: 5% auto;
```

```
background: #234465;
padding: 5px 10px 10px;
border-radius: 15px;
box-shadow: 5px 5px 10px #808080, 5px 5px 10px #6793c1 inset;
}

.currency-value {
    border: none;
    padding: 5px;
    border-radius: 5px;
}

.title {
    text-align: center;
}

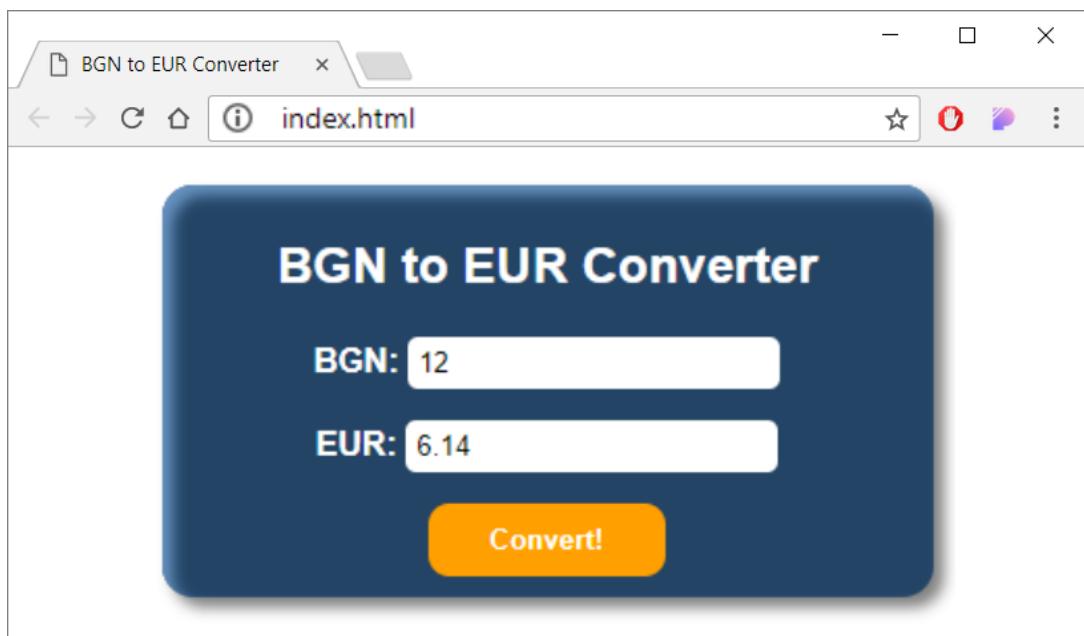
.item-currency {
    font-weight: 700;
}

.currency {
    margin: auto;
    padding-bottom: 15px;
}

.items {
    display: flex;
    flex-direction: column;
    justify-content: flex-start;
}

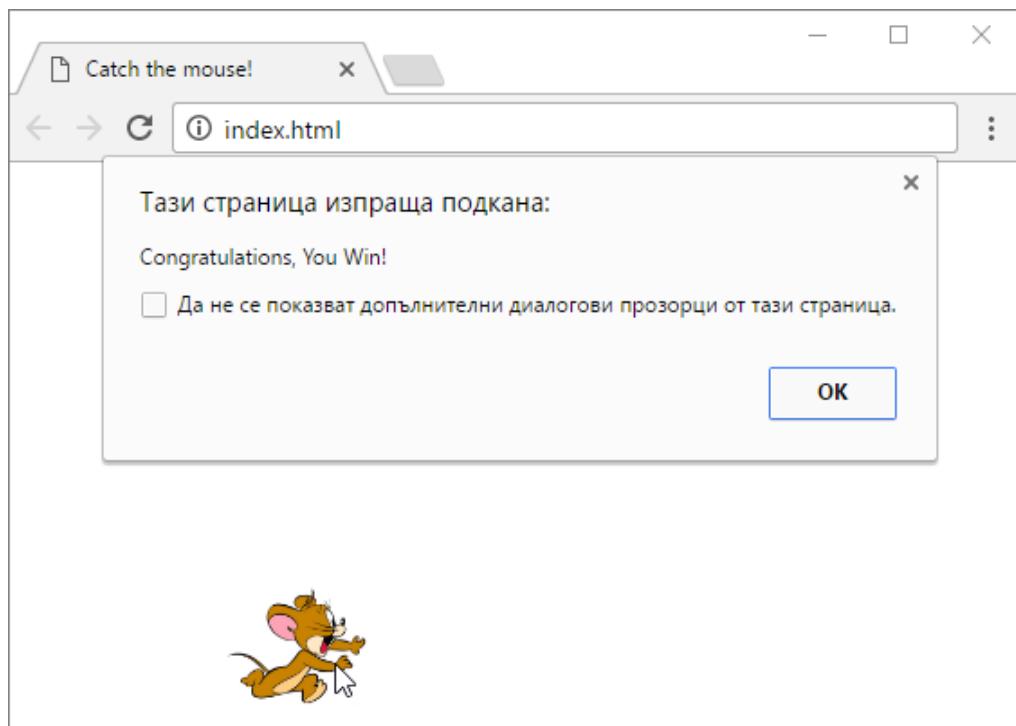
.primary-btn {
    margin: auto;
    border: none;
    padding: 10px 30px;
    border-radius: 10px;
    background-color: #ffa000;
    color: #FFFFFF;
    font-weight: 700;
}
```

6. Стартираме **index.html** файла и получаваме нещо такова (ако не сме объркали някъде):



### Уеб приложение: \*\*\* Хвани мишката!

При преместване на курсора на мишката върху изображението, то се премества на случайна позиция. Така се създава усещане, че „изображението бяга от курсора и е трудно да се хване“. При „хващане“ на изображението, се извежда съобщение-поздрав. Ето как би могло да изглежда готовото приложение:

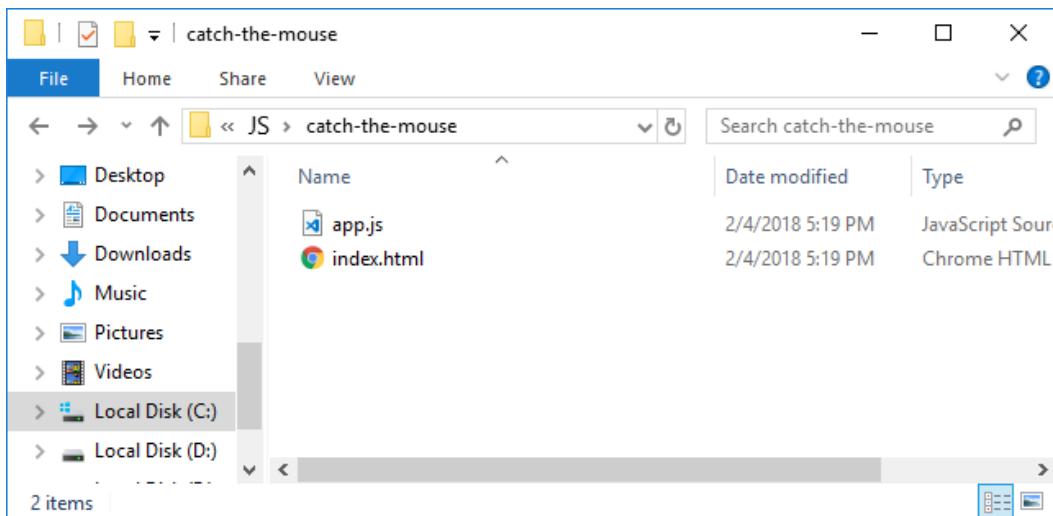


**Подсказка:** напишете обработчик за събитието **mouseover** и премествайте изображението на случайна позиция.

- Използвайте генератор за случайни числа **Math.random()**.
- Позицията на изображението се задава от свойството **style.position**.
- За да "хванете мишката", напишете функция за събитието **onclick**.

Ето и малко по-подробни насоки:

1. Създаваме нова папка **catch-the-mouse** в която ще съхраняваме файловете за уеб приложението.
2. В папката създаваме два файла: **index.html** и **app.js**. Структурата на папката трябва да изглежда по следния начин:



3. Можете да си помогнете с кода по-долу:

Файлът **index.html** трябва да изглежда по следния начин:

```
<!DOCTYPE html>
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
    <meta charset="utf-8" />
    <title>Catch the mouse!</title>
    <script src="app.js" type="text/javascript"></script>
</head>
<body>
    
</body>
</html>
```

Файлът **app.js** трябва да изглежда по следния начин:

```
function chaseMouse() {
```

```

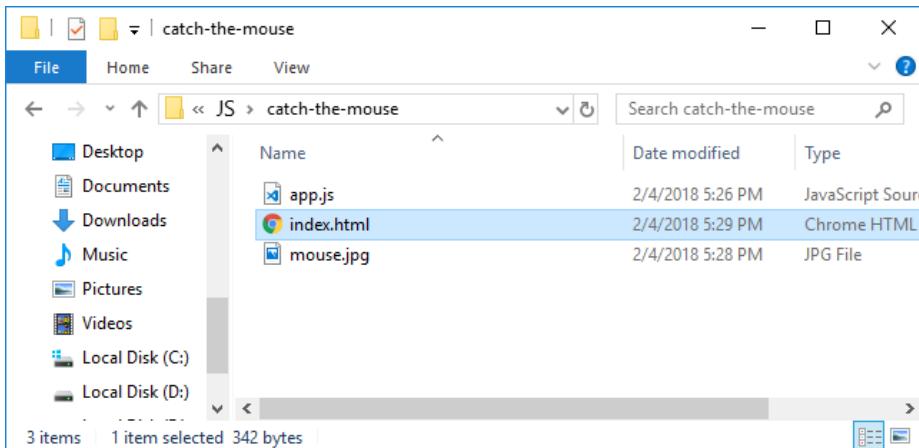
let img = document.getElementById("image");
img.style.position = "absolute";
img.style.left = (Math.random() * 300) + "px";
img.style.top = (Math.random() * 300) + "px";
}

function catchMouse() {
  alert("Congratulations, You Win!")
}

```

- Намираме изображение от интернет и го добавяме, като го именуваме **mouse.jpg**.

Тествайте приложението, като отворите папката на проекта в **explorer** и стартирате файла **index.html**:



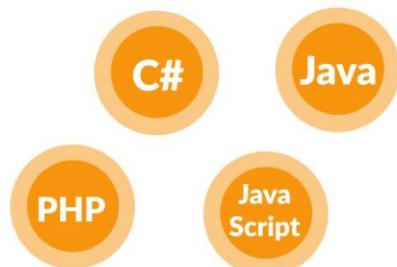
- Завършете приложението.

Ако имате трудности 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



**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](http://softuni.bg/apply)

# Глава 2.2. Прости пресмятания с числа – изпитни задачи

В предходната глава се запознахме с това как да подадем число на функция и как да отпечатаме резултат на конзолата. Разгледахме 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.

## Преговор: четене, извеждане на числа и пресмятания

Преди да преминем към задачите, да си припомним най-важното от изучавания материал в предходната тема. Ще започнем със създаването на функция, която прочита число.

### Четене на цяло число

Необходима ни е функция, на която се подава един аргумент **arg1**. В нея ще създадем променлива, в която да запазим числото (напр. **num**), в съчетание с метода **parseInt(...)**, който конвертира текст в число:

```
function readNumber(arg1) {  
    let num = parseInt(arg1);  
}
```

### Четене на дробно число

По същия начин, както четем цяло число, но този път ще използваме метода **parseFloat(...)**:

```
let num = parseFloat(arg1);
```

### Извеждане на текст по шаблон (placeholder)

**Placeholder** представлява израз, който ще бъде заменен с конкретна стойност при отпечатване. За да работи шаблонът, трябва да използваме наклонени кавички (апострофи) ``...``. Методът **console.log(...)** поддържа печтане на текст по шаблон, като аргументите, които трябва да отпечатаме, се задават в ``${...}``:

```
let firstName = "Ivan";  
let lastName = "Ivanov";  
let age = 19;  
let town = "Sofia";  
console.log(`You are ${firstName} ${lastName}, a ${age}-years  
old person from ${town}.`);
```

```
// You are Ivan Ivanov, a 19-years old person from Sofia.
```

## Аритметични оператори

Да си припомним основните аритметични оператори за пресмятания с числа.

### Оператор +

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

### Оператор -

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

### Оператор \*

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

### Оператор /

```
let result2 = 5 / 2; // резултатът е 2.5 (дробно деление)
```

## Конкатенация

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

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

## Изпитни задачи

Сега, след като си припомнихме как се четат и печатат числа на конзолата и как се извършват пресмятания с тях, можем да преминем 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 cm + място за стол и преминаване с размер 70 на 80 cm). **Коридорът** е широк поне 100 cm. Изчислено е, че заради **входната врата** (която е с отвор 160 cm) се губи точно 1 работно място, а заради **катедрата** (която е с размер 160 на 120 cm) се губят точно 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

Програмата чете 2 числа (аргумента), 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 отпечатат на конзолата едно цяло число: **броят места** в учебната зала.

## Примерен вход и изход

Вход	Изход	Чертеж
15 8.9	129	
8.4 5.2	39	

## Пояснения към примерите

В първия пример залата е дълга 1500 см. В нея могат да бъдат разположени **12 реда** ( $12 * 120 \text{ см} = 1440 + 60 \text{ см остатък}$ ). 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** ( $11 * 70 \text{ см} = 770 \text{ см} + 20 \text{ см остатък}$ ). **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** ( $7 * 120 \text{ см} = 840$ , 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** ( $6 * 70 \text{ см} = 420 \text{ см}$ , 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 дължината 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 дължината 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 дължината на едно работно място). Редиците ще намерим, като разделим **дължината на 120 см**. И при двете операции може да се получи **реално число** с цяла и дробна част, но **в променлива трябва да запазим 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, но тъй като в **JavaScript** има само един примитивен тип за числа (**Number**), това няма да е проблем. 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**.

## 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.

Първото, което трябва да направим, е да вземем входните данни. Създаваме си функция, на която се подават два аргумента. Запазваме техните стойности в две променливи, като използваме метода `parseFloat(...)` за преобразуване на зададената стрингова (текстова) стойност в дробно число:

```
function numberOfRows([arg1,arg2]) {
    let length = parseFloat(arg1);
    let width = parseFloat(arg2);
}
```

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 изпишем на **JavaScript**, 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 може да пробваме със следното търсене: "[JavaScript get whole number part of double](#)". Откриваме, че една от възможностите е да използваме метода **Math.trunc(...)**. Кодът по-долу е целенасочено замъглен и трябва да бъде довършен от читателя:

```
let length = Number(prompt("Length"));
let width = Number(prompt("Width"));

let area = length * width;
let integerArea = Math.trunc(area);
```

С **console.log(...)** отпечатваме резултата на конзолата:

## Тестване в Judge системата

Тествайте решението си: <https://judge.softuni.bg/Contests/Practice/Index/928#0>.

## Задача: зеленчукова борса

Градинар продава реколтата от градината си на зеленчуковата 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 приемем, че едно евро е равно на 1.94 лв.).

## Входни данни

На функцията се подават **4 аргумента**:

- Цена за килограм зеленчуци – число с плаваща запетая.
- Цена за килограм плодове – число с плаваща запетая.
- Общо килограми на зеленчуците – цяло число.
- Общо килограми на плодовете – цяло число.

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

## Изходни данни

Да се отпечатва 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

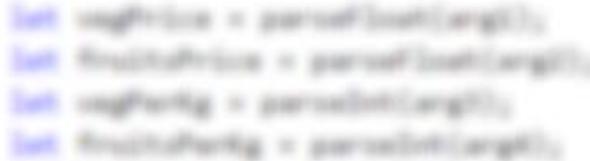
След като сме изяснили идеята си за решаването на задачата, можем да пристъпим към избора на подходящи типове данни. Да разгледаме **входа**: дадени са **две цели числа** за общия брой килограми на зеленчуците и плодовете, съответно променливите, които декларираме, за да пазим техните стойности, могат да бъдат конвертирани към число с **`parseInt(...)`**. За цените на плодовете и зеленчуците е указано, че ще бъдат подадени **две числа с плаваща запетая**, т.е. променливите ще бъдат преобразувани с метода **`parseFloat(...)`**.

## 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. С методите **`parseInt(...)`** и **`parseFloat(...)`** преобразуваме зададената текстова стойност съответно в цяло и дробно число.



Извършваме необходимите изчисления:

```
let vegTotal = vegPrice * vegPerKg;
let 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 **изчислим** 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.log(...)`** отпечатвате изхода на конзолата:

```
console.log((fruitTotal + vegTotal) / 1.94);
```

## Тестване в Judge системата

Тествайте решението си: <https://judge.softuni.bg/Contests/Practice/Index/928#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 плоочка се поставя за **0.2 минути**.

Напишете програма, която получава **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 плоочка 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

На функцията се подават **5 аргумента**:

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

#### Изходни данни

Да се отпечатат на конзолата **две числа**:

- броя **плочки**, необходим за ремонта
- **времето за поставяне**

Всяко число да бъде на нов ред и закръглено **до втория знак** след десетичната запетая.

#### Примерен вход и изход

Вход	Изход
20	
5	
4	19.9
1	3.98
2	

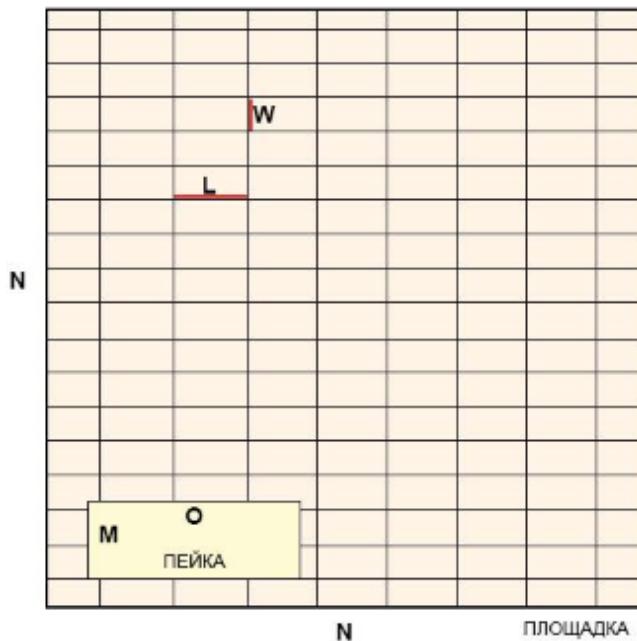
Вход	Изход
40	
0.8	
0.6	3302.08
3	660.42
5	

Обяснение към 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 по следния начин:



## 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 ѝ **N \* N**. 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 плочка 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, преобразувани с метода `parseInt(...)`**. За широчината и дължината на плочките ще ни бъдат подадени реални числа (с цяла и дробна част), затова за тях ще използваме **`parseFloat(...)`**.

## 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 разгледаме **входните данни** на задачата. Важно е да внимаваме за последователността, в която са дадени. Създаваме си необходимите променливи, в които да запазим входните данни, а с методите **`parseInt(...)`** и **`parseFloat(...)`** преобразуваме подадената стрингова стойност, съответно в цяло или дробно число:

```
// Ground length
let n = parseInt(arg1);
// Tile width
let w = parseFloat(arg2);
// Tile length
let l = parseFloat(arg3);
// Bench width
let m = parseInt(arg4);
// Bench length
let o = parseInt(arg5);
```

След като сме инициализирали променливите и сме запазили съответните стойности в тях, пристъпваме към **изчисленията**. Кодът по-долу е нарочно даден замъглен, за да може читателят да помисли самостоятелно над него:

```
let areaWithTiles = 100;
let tileArea = 25;
let tilesCount = areaWithTiles / tileArea;
let time = tilesCount * 0.2;

console.log(Math.round(tilesCount * 100) / 100);
console.log(Math.round(time * 100) / 100);
```

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**.

В условието на задачата е зададено закръгление на изхода **до втория знак след десетичната запетая**. Затова не можем просто отпечатаме стойностите с **console.log(...)**. Ще използваме метода **Math.round(...)**, който закръгля подаденото число до най-близкото цяло число. За да не загубим двата знака след запетаята, прилагаме метода върху числото умножено по 100, а после делим получения резултат на 100:

```
let tilesCount = areaWithTiles / tileArea;
let time = tilesCount * 0.2;

console.log(Math.round(tilesCount * 100) / 100);
console.log(Math.round(time * 100) / 100);
```

## Тестване в Judge системата

Тествайте решението си: <https://judge.softuni.bg/Contests/Practice/Index/928#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 процента** от крайната сума в евро.

## Входни данни

На функцията се подават 3 аргумента:

- **Броят биткойни** - цяло число в интервала [0 ... 20].
- **Броят китайски юани** - реално число в интервала [0.00 ... 50 000.00].
- **Комисионната** - реално число в интервала [0.00 ... 5.00].

## Изходни данни

На конзолата да се отпечата 1 число - резултатът от обмяната на валутите. Резултатът да се форматира до втората цифра след десетичния знак.

## Примерен вход и изход

Вход	Изход
1	
5	569.67
5	

Вход	Изход
20	
5678	12442.24
2.4	

Вход	Изход
7	
50200.12	10659.47
3	

Обяснение:

- 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**. Нея 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 преобразувана с метода **parseInt(...)**. Като брой **китайски юани** и **комисионна** ще получим **реално число**, следователно за тях използваме **parseFloat(...)**.

## 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 какви данни съдържат те. Инициализираме техните стойности: създаваме си променливи, в който да запазим подадените на функцията стрингови аргументи, като ги конвертираме към цяло или дробно число:

```
let bitcoins = parseInt(arg1);
let yuans = parseFloat(arg2);
let commission = parseFloat(arg3) / 100;
```

Извършваме необходимите изчисления:

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

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 `-=`, затова изразът **commission \* euro** се изпълнява първи, след което неговата стойност се изважда.

Накрая остава да изведем резултата на конзолата. Забелязваме, че се изисква форматиране на числената стойност до втория знак след десетичната точка. За разлика от предходната задача, тук дори и числото да е цяло, **трябва винаги да има два знака след десетичната точка** (например **5.00**). За целта можем да използваме метода `toFixed(...)`. С него можем да преобразуваме числото в текст, запазвайки определен брой знаци след десетичната запетая:

```
euro -=commission * euro;
console.log(euro.toFixed(2));
```

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 голям израз, в който директно след получаване на входните данни да сметнем изходната стойност. Например такъв израз би изглеждал ето така:

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

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

## Тестване в Judge системата

Тествайте решението си: <https://judge.softuni.bg/Contests/Practice/Index/928#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 изкаралото в България. Приема се, че в годината има точно 365 дни. Курсът на долара спрямо лева ще се подава на функцията.

### Входни данни

На функцията се подават 3 аргумента:

- Работни дни в месеца - цяло число в интервала [5 ... 30].
- Изкарани пари на ден - реално число в интервала [10.00 ... 2000.00].
- Курсът на долара спрямо лева /1 доллар = X лева/ - реално число в интервала [0.99 ... 1.99].

## Изходни данни

На конзолата да се отпечата едно 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:

- 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

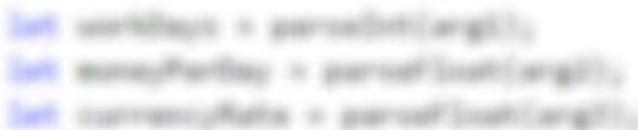
Работните дни за месец са дадени като **цяло число**, следователно за тяхната стойност може да декларираме променлива, в която да конвертираме до число с метода **`parseInt(...)`**. За изкараните пари, както и за курса на долара спрямо лева, ще получим **реално число**, следователно за тях използваме **`parseFloat(...)`**.

## 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**. Създаваме си променливи, в които запазваме подадените аргументи на функцията, като преобразуваме стринга към цяло или дробно число с **`parseInt(...)` / `parseFloat(...)`**:



Извършваме изчисленията:

```

let monthSalary = workDays * moneyPerDay;
let yearSalary = (monthSalary * 12) + (monthSalary * 2.5);
let taxes = yearSalary * 0.25;
let netSalary = yearSalary - taxes;
let 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 екрана. Забелязваме, че се **изиска форматиране на числената стойност до втория знак след десетичната точка**. Можем да използваме метода **`.toFixed(...)`** по същия начин като в предходната задача:

```
console.log((salaryInLeva / 365).toFixed(2));
```

## Тестване в Judge системата

Тествайте решението си: <https://judge.softuni.bg/Contests/Practice/Index/928#4>.

# Глава 3.1. Прости проверки

В настоящата глава ще разгледаме **условните конструкции в езика JavaScript**, 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 тук: <https://youtu.be/0GTknpt5mw8>.

## Оператори за сравнение

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.

Важно е да се отбележи, че в **JavaScript** се използват и още един вид оператори за **сравнение ==** и **различие !=**. Прилагането им без задълбочено разбиране води до проблеми и неочеквани резултати, затова няма да ги разглеждаме на този етап от нашата подготовка.

Повече информация за разликите между двата вида оператори за сравнение и различие можете да получите от тук: [https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Comparison\\_Operators](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Comparison_Operators)

## Примери за сравнение на числа

```
let a = 5;
let b = 10;
console.log(a < b); // True
console.log(a > 0); // True
console.log(a > 100); // False
```

```
console.log(a < a); // False
console.log(a <= 5); // True
console.log(a === 2); // False
```

## Примери за сравнение на променливи от тип "текст" (стринг)

```
let a = "book";
let b = "page";
console.log(a === b);      // False
console.log(a !== b);      // True
console.log(a === "book"); // True
console.log(a === "BooK"); // False
console.log(b !== "paGe"); // True
```

Важно е да се отбележи, че има значение дали буквите в нашият текст са **главни** или **малки**. Ако сравняваните стойности не са **напълно идентични**, резултатът, който ще получим винаги ще бъде **false**.

## Оператори за сравнение

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

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

## 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 като аргумент при извикване на функцията и проверяваме дали тя е отлична (**≥ 5.50**).

```
function isExcellent(n) {
    let grade = parseFloat(n);

    if (grade >= 5.50) {
        console.log("Excellent!");
    }
}
```

Тествайте кода от примера локално. 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!". Извикваме функцията като записваме нейното име, след което попълваме примерната стойност в скобите:

```
isExcellent(5.60); // Excellent!
```

### Тестване в Judge системата

Тествайте програмата от примера в Judge системата на СофтУни от следния линк: <https://judge.softuni.bg/Contests/Practice/Index/929#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 в езика **JavaScript** е следният:

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

## Пример: отлична оценка или не

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

```
function excellentOrNot(n) {
    let grade = parseFloat(n);

    if (grade >= 5.50) {
        console.log("Excellent!");
    } else {
        console.log("Not excellent.");
    }
}
```

## Тестване в Judge системата

Тествайте решението си: <https://judge.softuni.bg/Contests/Practice/Index/929#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 условния оператор. Когато искаме 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:

```
let color = "red";

if (color === "red")
    console.log("tomato");
else if (color === "yellow")
    console.log("banana");
console.log("bye");
```

Изпълнението на горния код ще изведе следния резултат на конзолата:

PROBLEMS	OUTPUT	DEBUG CONSOLE	TERMINAL
		Debugging with inspector protocol because Node.js node --inspect-brk=40193 test.js Debugger listening on ws://127.0.0.1:40193/3e60835	
		tomato bye	

С къдрави скоби кодът е по-ясен:

```
let color = "red";

if (color === "red") {
    console.log("tomato");
} else if (color === "yellow") {
    console.log("banana");
    console.log("bye");
}
```

На конзолата ще бъде отпечатано следното:

PROBLEMS    OUTPUT    DEBUG CONSOLE    TERMINAL

Debugging with inspector protocol because Node.js  
node --inspect-brk=48585 test.js  
Debugger listening on ws://127.0.0.1:48585/54729cd...  
tomato

## Пример: четно или нечетно

Да се напише функция, 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.

```
function isEven(arg1) {
    let num = parseInt(arg1);

    if (num % 2 === 0) {
        console.log("even");
    } else {
        console.log("odd");
    }
}
```

## Тестване в Judge системата

Тествайте решението си: <https://judge.softuni.bg/Contests/Practice/Index/929#2>.

## Пример: по-голямото число

Да се напише функция, която чете две цели числа и извежда по-голямото от тях.

Първата ни задача е да прочетем двете числа. След което, чрез прости **if-else** конструкции, в съчетание с **оператора за по-голямо (>)**, да направим проверка. Част от кода е замъглено умислено, за да могат читателите да изprobват наученото до момента.

```
function greaterNumber([arg1, arg2]) {
    let num1 = parseInt(arg1);
    let num2 = parseInt(arg2);

    if (num1 > arg2) {
        console.log(`The greater number is ${num1}`);
    } else {
        console.log(`The greater number is ${num2}`);
    }
}
```

## Тестване в Judge системата

Тествайте решението си: <https://judge.softuni.bg/Contests/Practice/Index/929#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 езикът **JavaScript** съществуват **два начина** за инициализиране на променливи. Чрез използването на ключовата дума **var** или **let**. Важно е да се отбележи разликата между тях, за да избегнем нежелани резултати при създаването на нашите функции.

Променливите, инициализирани чрез ключовата дума **var** имат свойствата на **глобални променливи**. Те се характеризират с това, че **могат да бъдат достъпвани навсякъде, независимо от това в коя част на нашия код са били декларириани**. При използването на ключовата дума **let**, нашата променлива приема **характеристиките на локална променлива**. Това означава, че животът ѝ започва от реда, в който сме я **дефинирали** и завършва до първата затваряща къдрава скоба **}** (на функцията, на **if** конструкцията и т.н.). Затова е важно да знаем, че всяка променлива, инициализирана с ключовата дума **let** вътре в тялото на **if**, **няма да бъде достъпна извън него**, освен ако не сме я дефинирали по-нагоре в кода.

В примера по-долу, на последните редове, ще се опитаме да извикаме дефинираните променливи. Ще успеем да отпечатаме **myMoney**, защото е декларирана в началото на нашата функция, преди **if** конструкцията, което я прави **достъпна навсякъде в функцията**. Също така ще е възможно да принтираме и **salary**, защото въпреки, че е декларирана в блока на **if** конструкцията, тя има характер на **глобална променлива** (понеже е дефинирана с **var**) и може да бъде

използвана навсякъде. При опитът за отпечатването на **bonus** променливата, която е инициализирана в **if** конструкцията, ще получим грешка, тъй като животът на тази променлива свършва с първата затваряща къдрава скоба **}**, която в случая е на **if** конструкцията:

```
let myMoney = 500;
let payDayDate = 07;
let todayDate = 10;

if (todayDate >= payDayDate) {
    var salary = 1000;
    let bonus = salary * 0.15;
    myMoney = myMoney + salary + bonus;
}

console.log(myMoney); // 1650
console.log(salary); // 1000
console.log(bonus); // Error
```

Използването на ключовата дума **var** за създаване на променливи е практика, която в миналото е била главният начин за дефиниране, но вече е непропоръчително да бъде прилагана. Затова и за всички примери в тази книга ще използваме ключовата дума **let**.

Важно е да се отбележи, че съществува и **трети начин** за инициализиране на променливи - чрез използването на ключовата дума **const**. Тези променливи имат същия обхват, както дефинираните чрез **let**, но имат една съществена разлика - приемат характеристиките на **константна променлива**. Това означава, че след първоначалното им инициализиране, тяхната стойност е **невъзможно да бъде променяна или предефинирана**.

## 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 else конструкция;
} else if (условие3) {
    // тяло на else 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 подава като параметър при извикване на функцията). Можем да прочетем числото и след това чрез **серия от проверки** отпечатваме съответстващата му английска дума:

```

function number1to9([arg1]) {
    let num = parseInt(arg1);

    if (num === 1) {
        console.log("one");
    } else if (num === 2) {
        console.log("two");
    } else if (num === 3) {
        console.log("three");
    }
    // TODO: add more checks
    else {
        console.log("number too big");
    }
}

```

Програмната логика от примера по-горе **последователно сравнява** входното число от функцията 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/929#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 функция, 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).

```
function scoreCalculator([arg1]) {
  let num = parseInt(arg1);
  let bonusScore = 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 ...
  }
}
```

```

        console.log("Bonus score: " + bonusScore);
        console.log("Total score: " + (num + bonusScore));
    }
}

```

Ето как би изглеждал резултатът при извикване на функцията с 175:

```

PROBLEMS      OUTPUT      DEBUG CONSOLE      TERMINAL

Debugging with inspector protocol because Node.js v
node --inspect-brk=12808 test.js
Debugger listening on ws://127.0.0.1:12808/233b6c78
Bonus score: 37
Total score: 212

```

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/929#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	Вход	Изход
35		22	
45	2:04	7	1:03
44		34	
50		14	
50	2:29	12	0:36
49		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, извеждаме водеща нула преди тях.

```
function sumSeconds([arg1, arg2, arg3]) {
  let firstCompetitor = parseInt(arg1);
  // TODO: Read also second and third competitor

  let seconds = firstCompetitor + secondCompetitor + thirdCompetitor
  let minutes = 0;

  if (seconds > 59) {
    minutes++;
    seconds = second - 60;
  }

  if (seconds > 59) {
    minutes++;
    seconds = second - 60
  }

  if (seconds < 10) {
    console.log(minutes + ":0" + seconds);
  } else {
    console.log(minutes + ":" + seconds);
  }
}
```

## Тестване в Judge системата

Тествайте решението си: <https://judge.softuni.bg/Contests/Practice/Index/929#6>.

## Задача: конвертор за мерни единици

Да се напише функция, 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)

Входна единица	Изходна единица
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 метода **toLowerCase()**, който ще направи всички букви малки. Както виждаме от таблицата в условието, можем да конвертираме само **между метри и никаква 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.

```
function metricConverter([arg1, arg2, arg3]) {
  let size = parseFloat(arg1);
  let sourceMetric = arg2.toLowerCase();
  let destMetric = arg3.toLowerCase();

  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.log(size + " " + destMetric);
}
```

## Тестване в Judge системата

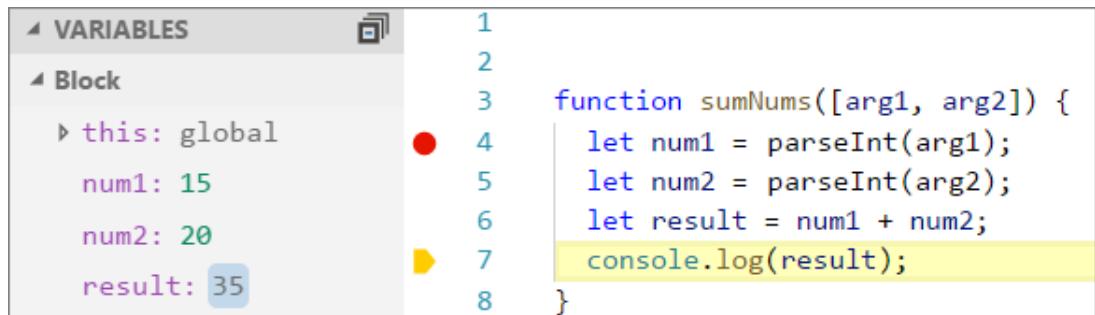
Тествайте решението си: <https://judge.softuni.bg/Contests/Practice/Index/929#7>.

## 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 проследим поетапно процеса на изпълнение. 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 Code

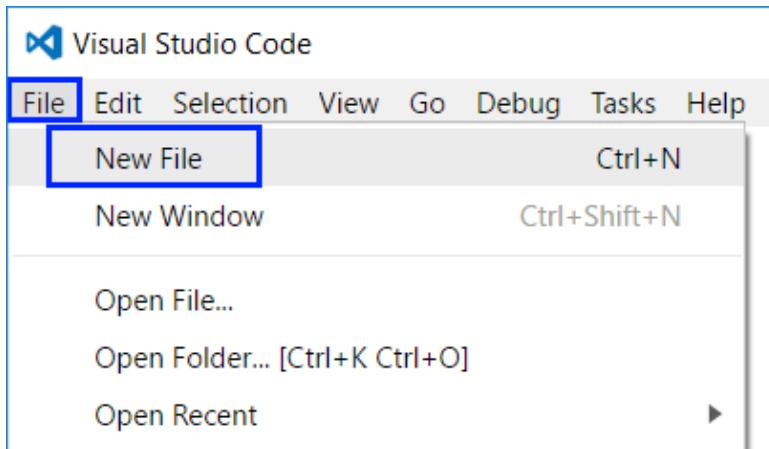
Добавяме точка, до която програмата да спре изпълнението си (**breakpoint**) и след това стартираме програмата в **debug режим** чрез натискане на бутона [F5]. Програмата се изпълнява до моментът, в който достигне нашата точка на прекъсване. След това преминаваме към **следващия ред** на изпълнение с [F10].

## Упражнения: прости проверки

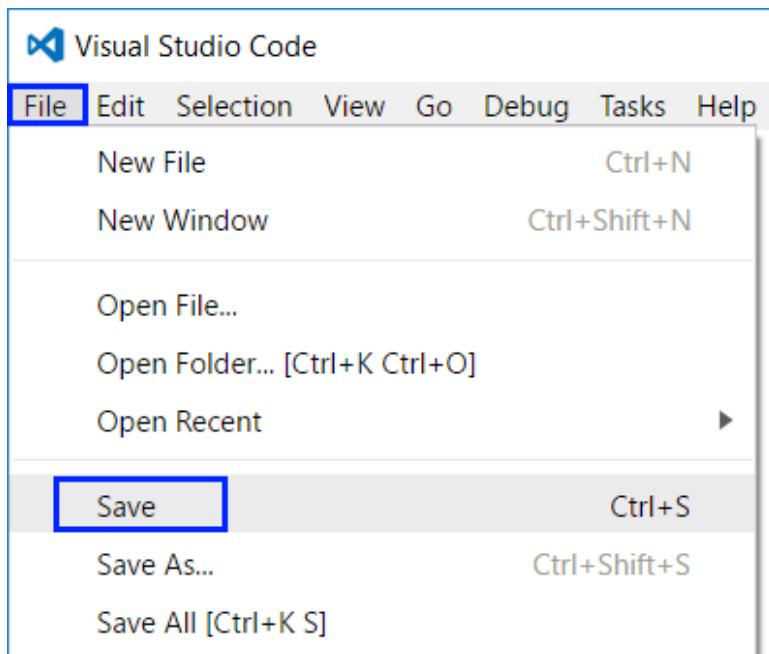
Нека затвърдим наученото в тази глава с няколко задачи.

### Празен Visual Studio Code файл

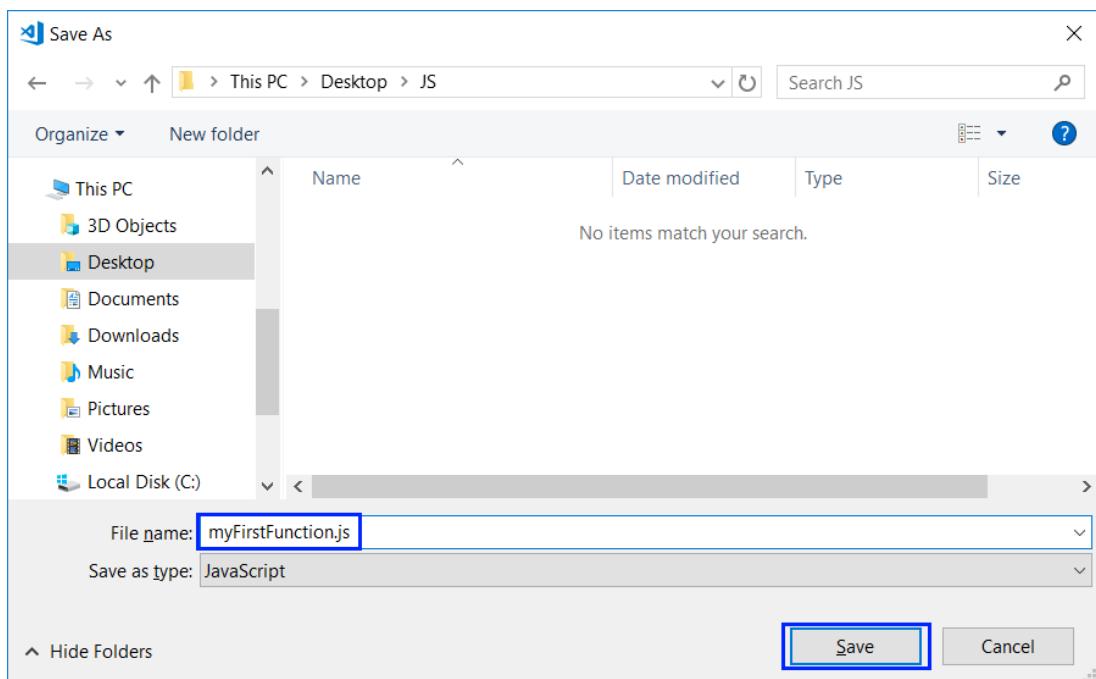
Стартираме Visual Studio Code. Създаваме нов файл [File] -> [New File]:



След това ни се появява нов файл, който за момента е анонимен за нашата система. За да може нашият код да бъде правилно разпознаваем е нужно да го запазим като JavaScript файл: [File] -> [Save]:



След това ни се отваря прозорец, в който трябва да зададем име на нашия файл, задължително с разширение **.js**:



## Задача: проверка за отлична оценка

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

### Примерен вход и изход

Вход	Изход	Вход	Изход	Вход	Изход
6	Excellent!	5.5	Excellent!	5.49	(няма изход)

### Насоки и подсказки

Създаваме нов анонимен файл чрез [File] -> [New File]. След това го запаметяваме ([File] -> [Save]), като **JavaScript файл**, като го запазим под разширение **.js**.

Вече имаме готов JavaScript файл. Остава да решим задачата. За целта пишем следния код:

```
function isExcellent(n) {
    let grade = parseFloat(n);
    if (grade >= 5.50) {
        console.log("Excellent!");
    }
}
```

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

```
isExcellent(5.60);
```

PROBLEMS	OUTPUT	DEBUG CONSOLE	TERMINAL
		Debugging with inspector protocol because Node.js v node --inspect-brk=12354 test.js Debugger listening on ws://127.0.0.1:12354/52b764e5	Excellent!

При стойности над 5.50 - получаваме резултат Excellent!.

```
isExcellent(5);
```

PROBLEMS	OUTPUT	DEBUG CONSOLE	TERMINAL
		Debugging with inspector protocol because Node.js node --inspect-brk=44626 test.js Debugger listening on ws://127.0.0.1:44626/8af48e3 Debugger attached. Waiting for the debugger to disconnect...	

При стойности под 5.50 - не получаваме резултат.

## Тестване в Judge системата

Тествайте решението си: <https://judge.softuni.bg/Contests/Practice/Index/929#0>.

### Excellent Result

```
1 function isExcellent(arg1) {  
2     let grade = parseFloat(arg1);  
3     if (grade >= 5.50) {  
4         console.log("Excellent!");  
5     }  
6 }
```

Allowed working time: 0.100 sec.      Allowed memory: 16.00 MB      Size limit: 16.00 KB      Checker: Case-Insensitive

JavaScript code (Node.js)

Points	Time and memory used	Submission date
100 / 100	Memory: 11.21 MB Time: 0.056 s	23:28:33 18.12.2017 <a href="#">Details</a>

## Задача: отлична оценка или не

Следващата задача от тази тема е да се напише JavaScript функция, която приема 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	Вход	Изход	Вход	Изход
6	Excellent!	5	Not Excellent!	5.49	Not excellent.

### Насоки и подсказки

Първо създаваме нов JavaScript файл. Следва да напишем кода на програмата. Може да си помогнем със следния примерен код:

```
function excellentOrNot(n) {
    let grade = parseFloat(n);

    if (grade >= 5.50) {
        console.log("Excellent!");
    } else {
        console.log("Not excellent.");
    }
}
```

Следва да извикваме функцията, като и подаваме примерни параметри и я тестваме дали работи коректно:

```
excellentOrNot(4.25);
excellentOrNot(5.60);
```

```
PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL
Debugging with inspector protocol because Node.js v
node --inspect-brk=3536 test.js
Debugger listening on ws://127.0.0.1:3536/f4ffcc2a-
Not excellent.
Excellent!
```

## Тестване в Judge системата

Тествайте решението си: <https://judge.softuni.bg/Contests/Practice/Index/929#1>.

Points	Time and memory used	Submission date	
100 / 100	Memory: 11.20 MB Time: 0.025 s	15:42:51 19.12.2017	<a href="#">Details</a>

## 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	Вход	Изход	Вход	Изход	Вход	Изход
2	even	3	odd	25	odd	1024	even

### Насоки и подсказки

Отново, първо добавяме **нов JavaScript файл**. 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: **let isEven = (num % 2 == 0)**.

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

```
isEven(42);

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL

Debugging with inspector protocol because Node.js v
node --inspect-brk=20036 test.js
Debugger listening on ws://127.0.0.1:20036/ee90abfa
even
```

## Тестване в Judge системата

Тествайте решението си: <https://judge.softuni.bg/Contests/Practice/Index/929#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	Вход	Изход	Вход	Изход	Вход	Изход
5 3	5	3 5	5	10 10	10	-5 5	5

## Насоки и подсказки

Както обикновено, първо трябва да добавим нов JavaScript файл. За кода на програмата ни е необходима единична **if-else** конструкция. Може да си помогнете частично с кода от картинката, който е умышлено замъглен, за да помисли читателя как да го допише сам:

```
function greaterNumber([arg1, arg2]) {
    let num1 = parseInt(arg1);
    let num2 = parseInt(arg2);

    if (num1 > arg2) {
        console.log("Greater number: " + num1);
    } else {
        console.log("Greater number: " + arg2);
    }
}
```

След като сме готови с имплементацията на решението, извикваме функцията като ѝ подаваме примерни параметри, **стартираме** програмата с [Ctrl+F5] и я тестваме:

```
greaterNumber(["5","3"]);

PROBLEMS OUTPUT DEBUG CONSOLE

Debugging with inspector protocol because node --inspect-brk=20387 test.js
Debugger listening on ws://127.0.0.1:20387
Greater number: 5
```

## Тестване в Judge системата

Тествайте решението си: <https://judge.softuni.bg/Contests/Practice/Index/929#3>.

## Задача: изписване на число до 9 с думи

Да се напише функция, която приема **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/929#4>.

### Задача: познай паролата

Да се напише функция, която **приема парола** (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** конструкцията.

### Тестване в Judge системата

Тествайте решението си: <https://judge.softuni.bg/Contests/Practice/Index/929#8>.

### Задача: число от 100 до 200

Да се напише функция, която като параметър **приема 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/929#9>.

## Задача: еднакви думи

Да се напише функция, която като параметър приема 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, е препоръчително да ги обърнете в долен регистър, за да не оказва влияние размера на буквите (главни / малки): `word = word.toLowerCase()`.

## Тестване в Judge системата

Тествайте решението си: <https://judge.softuni.bg/Contests/Practice/Index/929#10>.

## Задача: информация за скоростта

Да се напише функция, която като параметър приема 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/929#11>.

## Задача: лица на фигури

Да се напише функция, която приема размерите на геометрична фигура и пресмята лицето ѝ. Фигурите са четири вида: квадрат (**square**), правоъгълник (**rectangle**), кръг (**circle**) и триъгълник (**triangle**).

Като първи аргумент на функцията се подава вида на фигурата (**square**, **rectangle**, **circle**, **triangle**).

- Ако фигурата е **квадрат**, като следващ аргумент подаваме едно число – дължина на страната му.
- Ако фигурата е **правоъгълник**, като следващи аргументи подаваме две числа – дълчините на страните му.
- Ако фигурата е **кръг**, като следващ аргумент подаваме едно число – радиуса на кръга.
- Ако фигурата е **триъгълник**, като следващи аргументи подаваме 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/929#12>.

## Задача: време + 15 минути

Да се напише функция, която като параметър приема час и минути от 24-часово денонощие и изчислява колко ще е часът 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 и с **водеща нула**, когато е необходимо.

### 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/929#13>.

### Задача: еднакви 3 числа

Да се напише функция, в която се подават като аргументи 3 числа и се отпечатва дали те са еднакви (yes / no).

### Примерен вход и изход

Вход	Изход	Вход	Изход	Вход	Изход
5		5		1	
5	yes	4	no	2	
5		5		3	

### Тестване в Judge системата

Тествайте решението си: <https://judge.softuni.bg/Contests/Practice/Index/929#14>.

### Задача\*: изписване на число от 0 до 100 с думи

Да се напише функция, която превръща число в диапазона [0 ... 100] в текст.

### Примерен вход и изход

Вход	Изход	Вход	Изход	Вход	Изход
25	twenty five	42	forty two	6	six

### Насоки и подсказки

Проверете първо за **едноцифриeni числа** и ако числото е едноцифreno, отпечатайте съответната дума за него. След това проверете за **двуцифриени числа**. Тях отпечатвате 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/929#15>.

## Графично уеб приложение

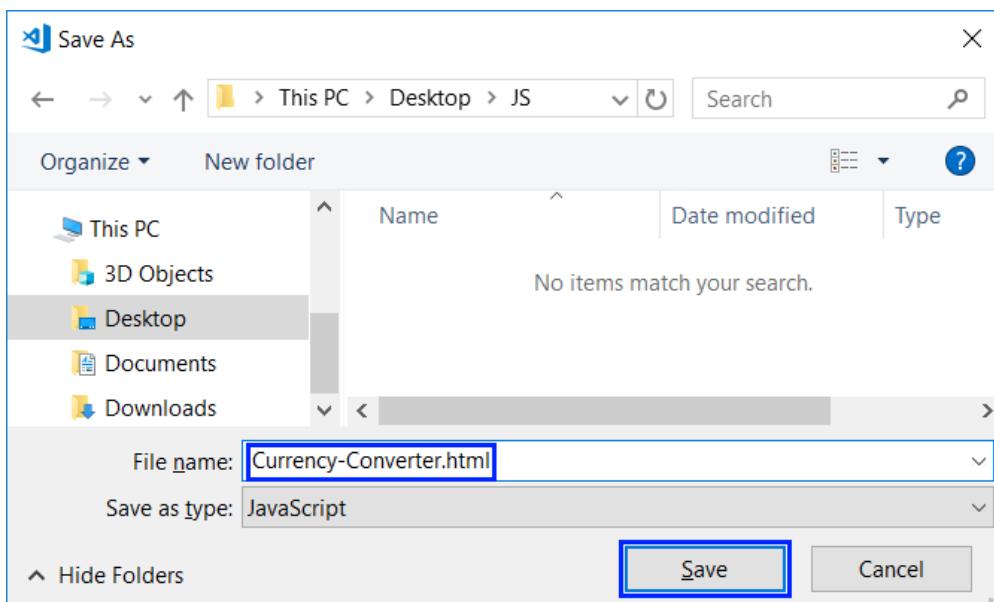
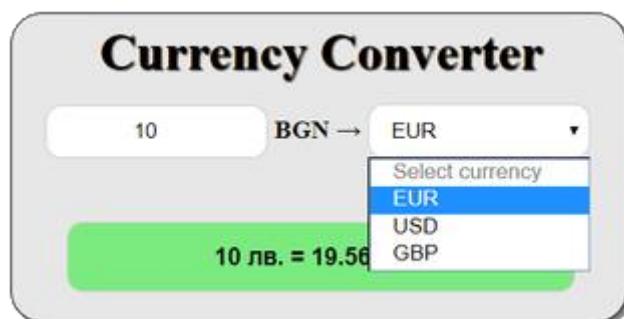
След като направихме няколко упражнения върху **условни конструкции (проверки)**, сега нека направим нещо по-интересно: приложение с графичен уеб потребителски интерфейс за конвертиране на валути. Ще използваме знанията от тази глава, за да избираме измежду няколко налични валути и съответно да извършваме 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.

За визуализация ще използваме **уеб браузър**, който визуализира **HTML** страници. Ще създадем нова такава и ще изградим **структурата, облика и функционалността** на нашето приложение.

Както обикновено **създаваме нов файл**, след това го запаметяваме с име **Currency-Converter**, но този път добавяме разширение **.html**.



Отваряме новосъздадения файл въвеждаме структурата на документа, под формата на HTML код:

```
<!DOCTYPE html>
<html>
<head>
    <meta charset="utf-8">
    <meta name="viewport" content="width=device-width, initial-
        scale=1">
    <meta http-equiv="x-ua-compatible" content="ie=edge">
    <title>Currency Converter</title>
    <style>

        /* enter CSS styling here */

    </style>
</head>
<body>
    <main id="converter-window">
        <h1>Currency Converter</h1>
        <form name="converter">
            <input type="number" placeholder="Enter number" min="0"
                id="cash-input" onkeyup="convert()" onchange="convert()">
            <span>BGN &#8594; </span>
            <select onchange="convert()" id="currency-options">
                <option selected disabled>Select currency</option>
                <option value="eur">EUR</option>
                <option value="usd">USD</option>
                <option value="gbp">GBP</option>
            </select>
            <br />
            <input type="text" name="result" id="result" disabled>
        </form>
    </main>
    <script>

        // Enter JavaScript functionality here

    </script>
</body>
</html>
```

Запазваме файла и го отваряме в уеб браузъра:



Вече сме изградили структурата на документа, но той може да бъде визуално подобрен след като добавим допълнителни **стилове**. За целта добавяме следният код в **<style>** секцията на нашия **HTML** документ:

```

body {
    background-color: #fff;
}

main {
    margin: 200px auto;
    height: 250px;
    width: 500px;
    background-color: #e7e7e7;
    border: 1px solid black;
    border-radius: 20px;
    box-shadow: 3px 3px 3px gray;
}

h1 {
    text-align: center;
    color: #000;
    text-shadow: 1px 1px 1px #000;
}

form {
    width: 400px;
    margin: 20px auto;
    text-align: center;
}

span {
    font-weight: bold;
    font-size: 16px;
}

input[type=number], input[type=text], select {

```

```

width: 140px;
padding: 8px 10px;
margin: 20px 0;
display: inline-block;
border: 1px solid #ccc;
border-radius: 10px;
box-sizing: border-box;
outline: none;
text-align: center
}
input[type=text] {
width: 80%;
margin-top: 20px;
background-color: #7beb80;
padding: 12px 10px;
color: black;
font-weight: bold;
font-size: 15px;
}
input:focus {
border: 2px solid #26a5e0;
}

```

Вече имаме и по-приятен изглед на нашето приложение. Остава да добавим и функционалността. Тя се добавя в `<script>` секцията в нашия HTML документ. Ще използваме следния JavaScript код за обработка на събитията:

```

function convert(){
let x = document.getElementById("cash-input").value;
let e = document.getElementById("currency-options");
let selected = e.options[e.selectedIndex].text;
let result;

if (selected === "EUR") {
    result = x + " " + "лв. = " + (x * 1.95583).toFixed(2)
        + " " + selected;
    document.getElementById("result").value = result;
} else if (selected === "USD") {
    result = x + " " + "лв. = " + (x * 1.63760).toFixed(2)
        + " " + selected;
    document.getElementById("result").value = result;
} else if (selected === "GBP") {
    result = x + " " + "лв. = " + (x * 2.22920).toFixed(2)
        + " " + selected;
    document.getElementById("result").value = result;
}

```

```
    }  
}
```

Горният код взима **сумата** за конвертиране от полето **cash-input** и избраната **валута** за резултата от полето **currency-options**. След това с **условна конструкция**, според избраната валута, сумата се дели на **валутния курс** (който е фиксиран твърдо в сорс кода). Накрая се генерира текстово **съобщение с резултата** (закръглен до 2 цифри след десетичния знак) и се записва в зелената кутийка **result**. Опитайте!

Ако имате проблеми с примерите по-горе, **гледайте видеото** в началото на тази глава или питайте във **форума на СофтУни**: <https://softuni.bg/forum>.

# Глава 3.2. Прости проверки – изпитни задачи

В предходната глава разглеждахме условните конструкции в езика JavaScript, 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 разгледаме някои задачи, давани на изпити. За целта нека първо си припомним конструкцията на логическата проверка:

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

**if** проверките се състоят от:

- **if** клауза
- булев израз - променлива от булев тип (**Boolean**) 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

Програмата чета **два реда** (аргумента):

- Първият ред (аргумент) съдържа числото **n** – брой километри – цяло число в интервала [1 ... 5000].
- Вторият ред (аргумент) 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, първата част от решението ще съдържа декларирането и инициализирането на **двете променливи**, в които ще пазим **стойностите на входните данни**:

```
function transportPrice([distance, dayOrNight]) {
```

Преди да започнем проверките е нужно да **декларираме** още една **променлива**, в която ще пазим **цената за транспорт**:

```
let price = 0;
```

## Извършване на проверки и съответните изчисления

След като вече сме **декларирали и инициализирали** променливите за входните данни, както и променливата, в която ще пазим цената, трябва да преценим **кои условия** от задачата да бъдат проверени първи.

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**:

```
let taxiRate = 0;
```

За да изчислим тарифата на таксито, ще използваме проверка от типа **if-else**:

```
if (dayOrNight === "day") {  
    taxiRate = 0.79;  
} else {  
    taxiRate = 0.90;  
}
```

След като е направено и това, вече може да пристъпим към изчислението на самата **цена за транспорта**. Ограниченията, които присъстват в условието на задачата, са относно **разстоянието**, което студента иска да пропътува. По тази причина, ще използваме **if-else** конструкция, с чиято помощ ще открием цената за транспорта в зависимост от подадените километри:

```
if (distance < 20) {  
    price = 0.70 + (distance * taxiRate);  
} else if (distance < 100) {  
    price = distance * 0.09;  
} else {  
    price = distance * 0.09;  
}
```

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 условието на проверката е **вярно** (връща **true**), на променливата, която пази стойността на цената на транспорта (**price**), ще **присвоим** съответната стойност. Тази стойност е равна на **първоначалната такса**, която **събираме** с неговата **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 транспортно 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** конструкцията, на променливата за цената на транспорта 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.log(price);
```

## Тестване в Judge системата

Тествайте решението си: <https://judge.softuni.bg/Contests/Practice/Index/930#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

На функцията се подават **четири числа** (аргумента):

- Първият ред (аргумент) съдържа числото **V** – **обем на басейна в литри** – цяло число в интервала [1 ... 10000].
- Вторият ред (аргумент) съдържа числото **P1** – **дебит на първата тръба за час** – цяло число в интервала [1 ... 5000].
- Третият ред (аргумент) съдържа числото **P2** – **дебит на втората тръба за час** – цяло число в интервала [1 ... 5000].
- Четвъртият ред (аргумент) 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

Първата ни стъпка е да прочетем входните данни:

```
function poolPipes([volume, pipe1, pipe2, 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:

```
let water = pipe1 * hours + pipe2 * hours
```

### 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 приста **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 **placeholders** и тъй като има условие **результатата в проценти** да се форматира до **две цифри** след **десетичния** знак **без закръгляне**, то за целта ще използваме метода **Math.trunc(...)**:

```

if (water <= volume) {
    console.log(
        `The pool is ${Math.trunc((water / volume * 100))}% full.
        Pipe 1: ${Math.trunc((water * 100 / volume * 100))}|.
        Pipe 2: ${Math.trunc((water * 100 / volume * 100))}|.
    );
} else {
    console.log(
        `For ${hours} hours the pool overflows
        with ${water - volume} liters.
    );
}

```

Ако проверката обаче върне резултат **false**, то това означава, че **количеството вода е по-голямо от обема** на 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/930#1>.

## Задача: поспаливатата 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

Програмата прочита едно цяло число (аргумент) - **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** представляват едно цяло число в интервала [0 ... 365].

```
function sleepyCatTom([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**, спасявата котка трябва да си играе и в **работните** за стопанина му. Числото, което прочитаме от конзолата, е броя 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:

```
let 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).

```
let 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 запишим 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 по-малко 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:

```
let difference = Math.abs( 
```

~~`totalPlayMinutes - 30000`~~

```
);
```

~~`let hours =`~~ 
~~`let 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` клаузата трябва проверим дали времето за игра е по-голямо от 30 000 (нормата).

## Обработка на изходните данни

Какъвто и резултат 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.log("Tom will run away");`~~
 ~~`console.log(`${Math.floor(hours)} hours and``~~
 ~~`| ${Math.floor(minutes)} minutes more for play`);`~~
~~`}`~~
`else {`

```
console.log("Tom sleeps well");
console.log(` ${Math.floor(hours)} hours and
| ${Math.floor(minutes)} minutes less for play`);
}
```

## Тестване в Judge системата

Тествайте решението си: <https://judge.softuni.bg/Contests/Practice/Index/930#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, който програмата прочита се състои от **точно 4 реда** (аргумента):

- 1-ви ред (аргумент): X кв.м е лозето – цяло число в интервала [10 ... 5000].
- 2-ри ред (аргумент): Y грозде за един кв.м. – реално число в интервала [0.00 ... 10.00].
- 3-ти ред (аргумент): Z нужни литри вино – цяло число в интервала [10 ... 600].
- 4-ти ред (аргумент): 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-високото 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 прочетем входните данни:

```
function harvest(  
    [vineyardArea, grapePerSquare, neededLiters, 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 **количеството 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 **количеството вино в литри**, което ще се получи от тази реколта. За тази цел декларираме още една променлива, в която ще пазим това **количество**. От условието стигаме до извода, че за да го пресметнем, е нужно да **разделим** количеството грозде в кг на 2.5:

```
let harvestPerVine = vineyardArea * grapePerSquare * 0.4;  
let wine = harvestPerVine / 2.5;
```

### Извършване на проверки и обработка на изходните данни

Вече сме направили нужните пресмятания и следващата стъпка е да **роверим** 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 ѝ чрез placeholder.

На втория ред има изискване да разпечатаме резултатите, като ги закръглим към по-високото цяло число, което ще направим с метода **Math.ceil(...)**. 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.

```
if ( _____ ) {
    let wineLeft = _____;

    console.log(`Good harvest this year!
Total wine: ${Math.floor( _____ )} liters.`);

    console.log(`${Math.ceil( _____ )} liters left ->
${Math.ceil( _____ )} liters per person.`);
}
```

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.log(`It will be a tough winter!
More ${Math.floor( _____ )} liters wine needed.`)
}
```

## Тестване в Judge системата

Тествайте решението си: <https://judge.softuni.bg/Contests/Practice/Index/930#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. Проектът е важен за фирмата и всеки служител задължително 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

Програмата прочита **точно 3 реда** (аргумента):

- На **първия** ред (аргумент) са **необходимите часове** – цяло число в интервала [0 ... 200 000].
- На **втория** ред (аргумент) са **дните**, с които фирмата разполага – цяло число в интервала [0 ... 20 000].
- На **третия** ред (аргумент) е **броят на всички служители** – 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!99 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 прочетем **входните данни**.

```
function firm([projectHours, availableDays, workers]) {
```

## Помощни изчисления

Следващата стъпка е да изчислим **количество**та на **работните часове** като умножим работните дни по 8 (всеки ден се работи по 8 часа) с броя на работниците 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 служителите с 2 (възможните часове извънработно време), като това също се умножава по броя на дните, с които фирмата разполага. От условието на задачата виждаме, че има условие **часовете** да са **закръглени към по-ниско цяло число**, което ще направим с метода **Math.floor(...)**.

```
let workDays = workDays * overtimeHours;
let overtimeHours = overtimeHours * workers * 2;
let workHours = workDays * overtimeHours * 8;
let totalHours = Math.Floor(workHours + overtimeHours);
```

## 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**.

```
if ( totalHours >= neededHours ) {
    console.log(`Yes! ${totalHours - neededHours} hours left.`);
} else {
    console.log(`Not enough time! ${neededHours - totalHours} hours needed. `);
```

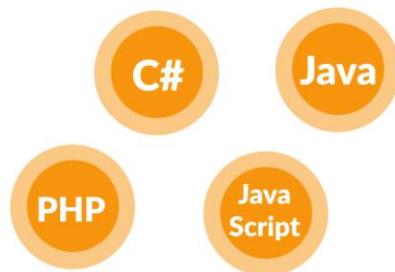
## Тестване в Judge системата

Тествайте решението си: <https://judge.softuni.bg/Contests/Practice/Index/930#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



**Софтууни** предоставя съвременно иновативно образование за програмиране, ИТ и дигитални умения на хиляди млади хора. Програмата "**Софтуерен университет**" изгражда истински **профессионалисти 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**, съдействайки на своите студенти за реализацията им като успешни софтуерни инженери.

### ПЪТЯТ НА СТУДЕНТА В СОФТУНИ



Кандидатствай

[softuni.bg/apply](http://softuni.bg/apply)

# Глава 4.1. По-сложни проверки

В настоящата глава ще разгледаме **вложените проверки** в езика JavaScript, 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 тук: [https://youtu.be/JRLA\\_zpQfpQ](https://youtu.be/JRLA_zpQfpQ).

## 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 избяга, най-вече чрез оптимизиране на структурата / алгоритъма на кода и/или чрез използването на друг вид условна конструкция, който ще разгледаме по-надолу в тази глава.

## Пример: обръщение според възраст и пол

Според въведени **възраст** (дробно число) и **пол (m / f)** да се отпечата обръщение:

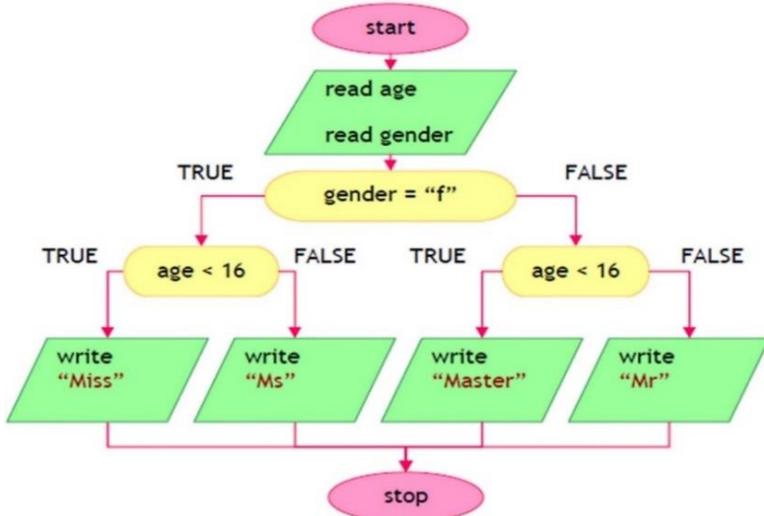
- "Mr." – мъж (пол "m") на 16 или повече години.
- "Master" – момче (пол "m") под 16 години.
- "Ms." – жена (пол "f") на 16 или повече години.
- "Miss" – момиче (пол "f") под 16 години.

## 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 **резултата** на първия ще определим кои от **другите** да изпълним.



След прочитане на входните данни от конзолата ще тряба да се изпълни следната примерна програмна логика:

```

function personalTitles([arg1, arg2]) {
  let age = Number(arg1);
  let gender = arg2;

  if (age < 16) {
    if (gender === 'm') {
      console.log("Master");
    } else if (gender === 'f') {
      console.log("Miss");
    }
  } else {
    if (gender === 'm') {
      console.log("Mr.");
    } else if (gender === 'f') {
      console.log("Ms.");
    }
  }
}
  
```

## Тестване в Judge системата

Тествайте решението си: <https://judge.softuni.bg/Contests/Practice/Index/931#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 **долен регистър** с метода **.toLowerCase()**, за да сравняваме продукти и градове без значение от малки / главни букви:

```
let product = arg1.toLowerCase();
let town = arg2.toLowerCase();
let quantity = Number(arg3);

if (town === "sofia") {
    if (product === "coffee" ) {
        console.log((0.50 * quantity).toFixed(2));
        //TODO: finish this...
    }
    else if (town === "plovdiv") //TODO: finish this...
    else if (town === "varna") //TODO: finish this...
```

## Тестване в Judge системата

Тествайте решението си: <https://judge.softuni.bg/Contests/Practice/Index/931#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 **неподредени** 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	<code>a &amp;&amp; b</code>
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**, **по-подреден** и **по-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:

```
let a = true;
let b = true;
let c = false;
let d = true;

let result = a && b && c && d;
// false (като d не се проверява)
```

Програмата ще се изпълни по **следния** начин: започва проверката от **a**, проверява **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 променливата има стойност **false**. След като програмата отчете, че аргументът **c** има стойност **false**, тя изчислява израза **до c**, **независимо** каква е стойността на **d**. Затова проверката на **d** се **прескача** и целият израз бива изчислен като **false**.

## 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  $x_1, y_1, x_2, y_2, x$  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	<p>A 2D coordinate system with x and y axes ranging from -5 to 12. A rectangle is drawn with vertices labeled <math>x_1, y_1</math> at (2, -3), <math>x_2, y_2</math> at (12, 3), and sides at <math>x=2</math> and <math>x=12</math>, <math>y=-3</math> and <math>y=3</math>. A point <math>(x, y)</math> is marked inside the rectangle at approximately (7, -1).</p>

### 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.

```
let x1 = Number(arg1);
let y1 = Number(arg2);
let x2 = Number(arg3);
let y2 = Number(arg4);
```

```

let x = Number(arg5);
let y = Number(arg6);

if (x >= x1 && x <= x2 && y >= y1 && y <= y2) {
    console.log("Inside");
} else {
    console.log("Outside")
}

```

## Тестване в Judge системата

Тествайте решението си: <https://judge.softuni.bg/Contests/Practice/Index/931#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	a    b
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 **роверява** 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:

```

let a = false;
let b = true;
let c = false;

```

```
let d = true;
let 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
kiwi	fruit
java	unknown

### Решение

Трябва да използваме няколко условни проверки с логическо "ИЛИ" (`||`):

```
let s = arg1;

if (s === "banana" || s === "apple" || s === "kiwi" ||
    s === "cherry" || s === "lemon" || s === "grapes") {
    console.log("fruit");
} else if (s === "tomato" || s === "cucumber"
           || s === "pepper" || s === "carrot") {
    console.log("vegetable");
} else {
    console.log("unknown");
}
```

### Тестване в Judge системата

Тествайте решението си: <https://judge.softuni.bg/Contests/Practice/Index/931#3>.

## Логическо отрицание

Логическо отрицание (оператор `!`) означава да **не е изпълнено** дадено условие.

a	<code>!a</code>
<code>true</code>	<code>false</code>

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	<code>invalid</code>
150	(няма изход)
220	<code>invalid</code>

#### Решение

```
let num = Number(arg1);
let inRange = (num >= 100 && num <= 200) || num === 0;

if (!inRange) {
    console.log("invalid");
}
```

#### Тестване в Judge системата

Тествайте решението си: <https://judge.softuni.bg/Contests/Practice/Index/931#4>.

## Операторът скоби ()

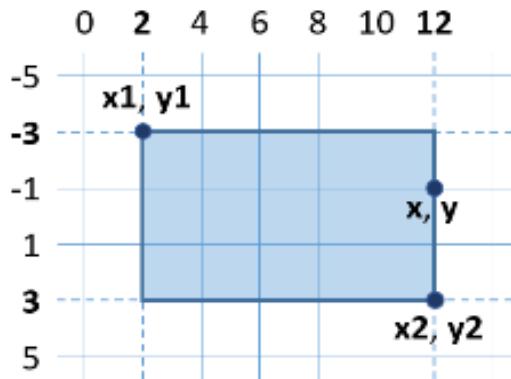
Както останалите оператори в програмирането, така и операторите `&&` и `||` имат приоритет, като в случая `&&` е с по-голям приоритет от `||`. Операторът `()` служи за **промяна на приоритета на операторите** и се изчислява пръв, също както в математиката. Използването на скоби също така придава по-добра 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.log("Border");
}
```

Предходната проверка може да се опрости по този начин:

```

let onLeftSide = (x === x1) && (y >= y1) && (y <= y2);
let onRightSide = (x === x2) && (y >= y1) && (y <= y2);
let onLowerSide = (y === y1) && (x >= x1) && (x <= x2);
let onUpperSide = (y === y2) && (x >= x1) && (x <= x2);

if (onLeftSide || onRightSide || onLowerSide || onUpperSide) {
    console.log("Border");
}

```

Вторият начин с допълнителните булеви променливи е по-дълъг, но е много по-разбираем от първия, нали? Препоръчително е, когато пишем булеви условия, да ги правим **лесни за четене и разбиране**, а не кратки. Ако се налага, ползваме допълнителни променливи със смислени имена. Имената на булевите променливи трябва да подсказват каква стойност се съхранява в тях.

Остава да допишете кода, за да отпечатва “**Inside / Outside**”, ако точката не е върху някоя от страните на правоъгълника.

## Тестване в Judge системата

След като допишете решението, може да го тествате тук: <https://judge.softuni.bg/Contests/Practice/Index/931#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 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 имена плод да се отпечата “**error**”.

## Примерен вход и изход

Вход	Изход	Вход	Изход	Вход	Изход	Вход	Изход
orange Sunday 3	2.70	kiwi Monday 2.5	6.75	grapes Saturday 0.5	2.10	tomato Monday 0.5	error

## Решение

```

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 в Judge системата

Тествайте решението си: <https://judge.softuni.bg/Contests/Practice/Index/931#6>.

## Пример: търговски комисионни

Фирма дава следните комисионни на търговците си според града, в който работят и обема на продажбите s:

Град	0 <= s <= 500	500 < s <= 1000	1000 < s <= 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 метода `.toLowerCase()`). Първоначално задаваме комисационната да е `-1`. Тя ще бъде променена, ако градът и ценовият диапазон бъдат намерени в таблицата с комисционните. За да изчислим комисационната според града и обема на продажбите се нуждаем от няколко вложени `if` проверки, както е в примерния код по-долу:

```
let commission = -1;

if (town === "sofia") {
    if (sells >= 0 && sells <= 500) commission = 0.05;
    else if (sells > 500 && sells <= 1000) commission = 0.07;
    // TODO: check the other price range ...
}

else if (town === "varna") // TODO: check the price range ...
else if (town === "plovdiv") // TODO: check the price range ...

if (commission >= 0) {
    commission = sells * commission;
    console.log(commission.toFixed(2));
} else console.log("error");
```

## Тестване в Judge системата

Тествайте решението си: <https://judge.softuni.bg/Contests/Practice/Index/931#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-case**. 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**. 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

### Решение

```
let num = Number(arg1);

switch (num) {
    case 1: console.log("Monday"); break;
    case 2: console.log("Tuesday"); break;
    ...
    case 7: console.log("Sunday"); break;
    default: console.log("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 в Judge системата

Тествайте решението си: <https://judge.softuni.bg/Contests/Practice/Index/931#8>.

## Множество етикети в switch-case

В JavaScript 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.log("mammal");
        break;
    case "crocodile":
    case "tortoise":
    case "snake":
        console.log("reptile");
        break;
    default:
        console.log("unknown");
        break;
}
```

## Тестване в Judge системата

Тествайте решението си: <https://judge.softuni.bg/Contests/Practice/Index/931#9>.

## Какво научихме от тази глава?

Да си припомним новите конструкции и програмни техники, с които се запознахме 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.log(...);
```

## 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	Вход	Изход
Premiere 10 12	1440.00 leva	Normal 21 13	2047.50 leva

## Насоки и подсказки

При прочитането на входа можем да обърнем типа на прожекцията в малки букви (с метода `.toLowerCase()`). Създаваме и инициализираме променлива, която ще ни съхранява изчислените приходи. В друга променлива пресмятаме пълния капацитет на залата. Използваме условната конструкция `switch-case`, за да изчислим прихода в зависимост от вида на прожекцията и отпечатваме резултата на конзолата в зададения формат (потърсете нужната `JavaScript` функционалност в интернет).

Примерен код (части от кода са замъглени с цел да се стимулира самостоятелно мислене и решение):

```
let projection = arg1.toLowerCase();
let rows =
let columns =
let full =
let income =
switch (projection) {
  case "premiere":
    income = 100 * 100;
    break;
  case "student":
    income = 50 * 100;
    break;
  case "discounted":
    income = 5 * 100;
    break;
}
console.log(`Projection: ${projection}, Rows: ${rows}, Columns: ${columns}, Income: ${income}`);
```

## Тестване в Judge системата

Тествайте решението си: <https://judge.softuni.bg/Contests/Practice/Index/931#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 уикенда, 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.  $2.15 \rightarrow 2$ ;  $9.95 \rightarrow 9$ ).

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 метода **.toLowerCase()**. Последователно пресмятаме **уикенда** 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 **JavaScript** клас с такава функционалност в интернет).

Примерен код (части от кода са замъглени с цел да се стимулира самостоятелно мислене и решение):

```
let year = arg1.toLowerCase();
let holidays =
let weekendsHome = ...

let sofiaWeekends = ...
let playSofia = ...
let playTotal = ...
```

```

if ( число < playTotal ) {
    playTotal = Math.floor(15 * playTotal) / число + playTotal;
}
else if ( число > playTotal ) {
    playTotal = Math.floor(playTotal);
}

console.log(playTotal);

```

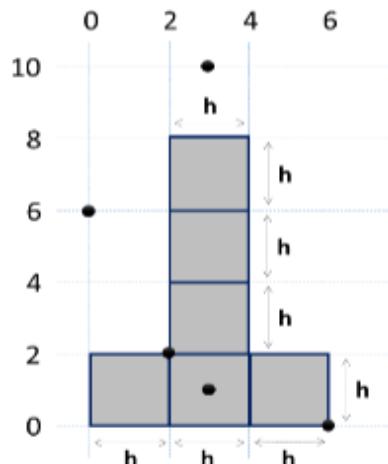
### Тестване в Judge системата

Тествайте решението си: <https://judge.softuni.bg/Contests/Practice/Index/931#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  $\{2*h, 4*h\}$ . 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.
- 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**).

Примерен код (части от кода са замъгленi с цел да се стимулира самостоятелно мислене и решение):

```
let h = Number(prompt());
let x = Number(prompt());
let y = Number(prompt());

let outRectangle1 = ((x < 0) || (x > 3 * h)) || ((y < 0) || (y > h));
let outRectangle2 = ((x < h) || (x > 2 * h)) || ((y < h) || (y > 2 * h));

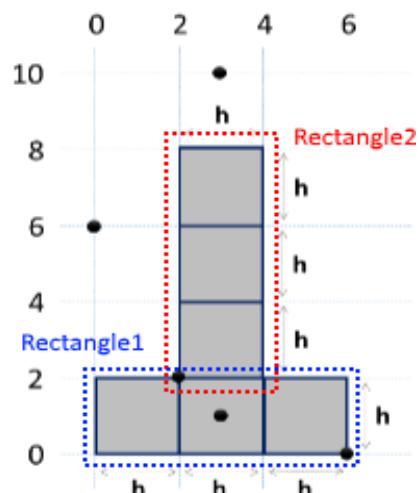
let inRectangle1 = ((x > 0) && (x < 3 * h)) && ((y > 0) && (y < h));
let inRectangle2 = ((x > h) && (x < 2 * h)) && ((y > h) && (y < 2 * h));

let commonBorder = (x > h) && (x < 3 * h) && (y > 0) && (y < h);

if (!outRectangle1 && !outRectangle2 && !commonBorder) {
    console.log("outside")
} else if (!outRectangle1 && !outRectangle2 && commonBorder) {
    console.log("inside");
} else {
    console.log("border");
}
```

### Тестване в Judge системата

Тествайте решението си: <https://judge.softuni.bg/Contests/Practice/Index/931#12>.

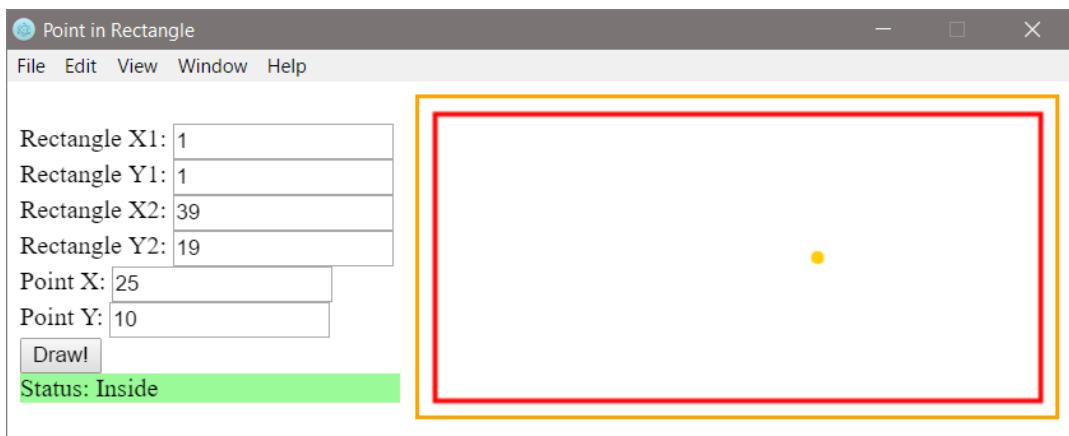


## Упражнение: графично приложение с по-сложни проверки

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.



**Внимание:** това приложение е значително **по-сложно** от предходните графични приложения, които разработвахме до сега, защото изисква ползване на функции за чертане (Canvas), работа с HTML, JavaScript и GUI framework (Electron).

Следват инструкции за изграждане на приложението стъпка по стъпка:

1. Първо ще си създадем отделна папка за проекта на нашето приложение с подходящо име, например "Point-in-Rectangle".
2. Инсталлираме **Electron** – работна рамка (**framework**) за създаване на графични (**GUI**) приложения с JavaScript. Изпълняваме следната команда на конзолата (Command Prompt / Bash):

```
npm install -g electron
```

3. В папката на проекта създаваме **JavaScript файл** с име **main.js** като във VS Code натиснем **[Ctrl + N]**. След това записваме новия файл с **[Ctrl + Shift + S]** и въвеждайки желаното име на файла.
4. Кодът, описан в **main.js**, управлява събитията и създава нови прозорци в приложението. Трябва да изглежда по следния начин:

```
const path = require('path');
const url = require('url');
const { app, BrowserWindow } = require('electron');

let win;

function createWindow () {
    win = new BrowserWindow({
        width: 750, height: 300, resizable: false});
    win.loadURL(url.format({
        pathname: path.join(__dirname, 'index.html'),
        protocol: 'file:',
        slashes: true
    }));
    win.on('closed', () => { win = null; });
}

app.on('ready', createWindow);

app.on('window-all-closed', () => { app.quit(); });

app.on('activate', () => {
    if (win === null) {
        createWindow();
    }
});
```

5. В папката на проекта създаваме и нов HTML файл с име **index.html**. Тагът **<title>** е задължителен за всеки **html** документ и дефинира заглавието му. Влизаме в него и написваме "**Point in Rectangle**":

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="UTF-8">
    <title>Point and Rectangle</title>
  </head>

  <body>

  </body>
</html>
```

Добавяме следния код под тага **<title>** в index.html файла:

```
<script src="app.js" type="text/javascript"></script>
```

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="UTF-8">
    <title>Point and Rectangle</title>
    <script src="app.js" type="text/javascript"></script>
  </head>

  <body>

  </body>
</html>
```

По този начин се осъществява връзката между файловете index.html и app.js (който ще създадем малко по-късно). Тагът **<body>** дефинира тялото на html документа. Написваме в него следния код:

```
<body>
  <div style="float:left">
    <br />
    <label>Rectangle X1:</label>
    <input id="rect-x1" type="number" />
    <br />

    <label>Rectangle Y1:</label>
    <input id="rect-y1" type="number" />
    <br />
```

```

<label>Rectangle X2:</label>
<input id="rect-x2" type="number" />
<br />

<label>Rectangle Y2:</label>
<input id="rect-y2" type="number" />
<br />

<label>Point X:</label>
<input id="point-x" type="number" />
<br />

<label>Point Y:</label>
<input id="point-y" type="number" />
<br />

<input type="button" onclick="draw()" value="Draw!" />
<br />

<div id="result">
    <label>Status:</label>
    <span id="status"></span>
</div>
</div>

<div style="float:right">
    <canvas style="border: 2px solid orange;" id="a" width="400" height="200">
        Sorry, your browser does not support HTML5 Canvas.
    </canvas>
</div>
</body>

```

За въвеждане координатите на правоъгълника и на точката, използваме **input** полета от тип **Number**, с тагове **<label>**. За да чертаем геометрични фигури в приложението, използваме html тага **<canvas>**:

```

<div style="float:right">
    <canvas style="border: 2px solid orange;" id="a"
        width="400" height="200">
        This text is displayed if your browser does not
        support HTML5 Canvas.
    </canvas>
</div>

```

Той приема следните параметри:

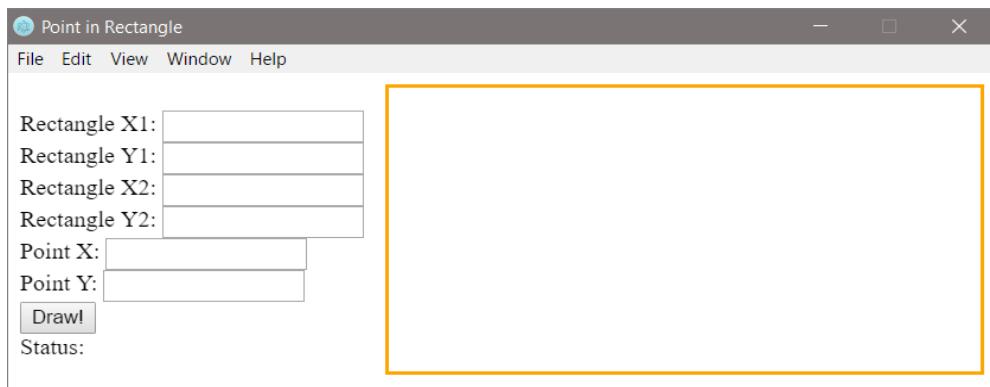
- Ширина (width) в пиксели (px)
- Височина (height) в пиксели (px)
- Очертание (border)

За да се отразяват промените в приложението, файловете трябва да се запазват с **[Ctrl+S]**.

За да стартираме приложението, изпълняваме в конзолата (в папката на текущия проект) следната команда:

```
electron .
```

Приложението трябва да изглежда по следния начин:



6. Остава да се имплементира най-сложната част: **визуализация на правоъгълника и точката** в полето на елемента **<canvas>** чрез функцията **draw()** във файла **app.js**, който създаваме в директорията на приложението, по начина, описан в Точка 2.

Създаваме **CanvasRenderingContext2D** обект като напишем следния код:

```
// Create canvas element
let canvas = document.getElementById('a');
let context = canvas.getContext('2d');
```

Елементът **<canvas>** е поле, в което обектът, генериран чрез метода **.getContext('2d')**, чертае графики, текст, изображения и други елементи. В случая променливата **context** представлява този обект. Записваме в отделни променливи координатите на двета ъгъла на правоъгълника:

```
// Get input for rectangle coordinates
let rectX1 =
    Number(document.getElementById("rect-x1").value) * 10;
let rectY1 =
    Number(document.getElementById("rect-y1").value) * 10;
```

```
let rectX2 =
    Number(document.getElementById("rect-x2").value) * 10;
let rectY2 =
    Number(document.getElementById("rect-y2").value) * 10;
```

Стойностите на координатите са достъпни чрез **id** на **<input>** полетата. За по-добра визуализация на екрана, мащабираме стойностите като **ги увеличаваме 10 пъти**. Следващата стъпка е да се пресметнат страните на правоъгълника, тъй като обектът **context** рисува правоъгълник по четири параметъра: **x** - координата, **y** - координата, **ширина** в пиксели и **височина** в пиксели:

```
// Calculate rectangle parameters
let rectWidth = Math.abs(rectX1 - rectX2);
let rectHeight = Math.abs(rectY1 - rectY2);
```

Можем да използваме кода по-долу, който рисува червен правоъгълник, според зададените във формата координати, използвайки метода [\*\*.strokeRect\(...\)\*\*](#):

```
// Set rectangle style
context.strokeStyle = "#ff0000";
context.lineWidth = 3;

// Draw rectangle with given parameters
context.strokeRect(rectX1, rectY1, rectWidth, rectHeight);
```

Аналогично на правоъгълника, взимаме координатите на точката и ги мащабираме. След това задаваме стил на точката - оранжев цвят. За по-добра визуализация на екрана, преобразуваме точката в кръг с метода [\*\*.arc\(...\)\*\*](#). Този метод приема пет параметъра: **x**-координата, **y**-координата, **радиус**, **начало на дъгата** в радиани, **край на дъгата** в радиани:

```
// Get input for point coordinates
let pointX =
    Number(document.getElementById("point-x").value) * 10;
let pointY =
    Number(document.getElementById("point-y").value) * 10;

// Set point style and draw point
context.beginPath();
context.fillStyle = "#ffcc00";
context.arc(pointX, pointY, 4, 0, 2 * Math.PI);
context.closePath();
context.fill();
```

За да отразим резултатите в **if** проверките, запазваме в отделни променливи следните елементи от html кода:

```
// Assign variables to (<div id="result">) and (<span id="status">) html elements
let result = document.getElementById("status");
let output = document.getElementById("result");
```

Последната стъпка е проверка на позицията на точката спрямо правоъгълника:

```
// Check point position
if () {
    result.innerHTML = "Inside";
    output.style.backgroundColor = "palegreen";
} else if () {
    result.innerHTML = "Border";
    output.style.backgroundColor = "gold";
} else {
    result.innerHTML = "Outside";
    output.style.backgroundColor = "lightsalmon";
}
```

Нека помислим как **да допишем** недовършените (нарочно) условия в **if** конструкциите. Кодът по-горе нарочно не се компилира, защото целта му е читателят да помисли как и защо работи и да допълни липсващите части. Горният код взима координатите на правоъгълника и точката и проверява дали точката е вътре, вън или на страна на правоъгълника. При визуализацията на резултата се сменя и цвета на фона на текстовия блок, който го съдържа.

Това е пълната версия на функцията **draw()**:

```
function draw() {
    // Create canvas element
    let canvas = document.getElementById('a');
    let context = canvas.getContext('2d');

    // Clear canvas window
    context.clearRect(0, 0, canvas.width, canvas.height);

    // Get input for rectangle coordinates
    let rectX1 =
        Number(document.getElementById("rect-x1").value) * 10;
    let rectY1 =
        Number(document.getElementById("rect-y1").value) * 10;
    let rectX2 =
        Number(document.getElementById("rect-x2").value) * 10;
    let rectY2 =
        Number(document.getElementById("rect-y2").value) * 10;
```

```
// Calculate rectangle parameters
let rectWidth = Math.abs(rectX1 - rectX2);
let rectHeight = Math.abs(rectY1 - rectY2);

// Set rectangle style
context.strokeStyle = "#ff0000";
context.lineWidth = 3;

// Draw rectangle with given parameters
context.strokeRect(rectX1, rectY1, rectWidth, rectHeight);

// Get input for point coordinates
let pointX =
    Number(document.getElementById("point-x").value) * 10;
let pointY =
    Number(document.getElementById("point-y").value) * 10;

// Set point style and draw point
context.beginPath();
context.fillStyle = "#ffcc00";
context.arc(pointX, pointY, 4, 0, 2 * Math.PI);
context.closePath();
context.fill();

// Assign variables to (div id="result") and (span
// id="status") html elements
let result = document.getElementById("status");
let output = document.getElementById("result");

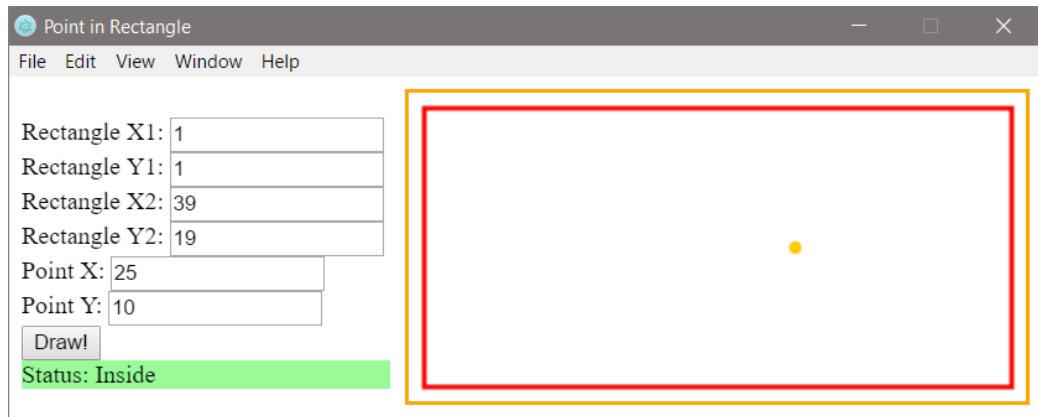
// Check point position
if (pointX > rectX1 && pointX < rectX2
    && pointY > rectY1 && pointY < rectY2) {
    result.innerHTML = "Inside";
    output.style.backgroundColor = "palegreen";
} else if ((pointX === rectX1 || pointX === rectX2)
    && pointY >= rectY1 && pointY <= rectY2
    || (pointY === rectY1 || pointY === rectY2)
    && pointX >= rectX1 && pointX <= rectX2) {
    result.innerHTML = "Border";
    output.style.backgroundColor = "gold";
} else {
    result.innerHTML = "Outside";
    output.style.backgroundColor = "lightsalmon";
}
}
```

Стартираме приложението чрез файла `index.html` и го тестваме (с въвеждане на различни входни данни). Пробваме да въвеждаме различни правоъгълници и да позиционираме точката на различни позиции, да преоразмеряваме приложението и виждаме дали се държи коректно. Ако приложението не работи коректно, проверяваме за грешки. Най-вероятната причина за грешка е, ако сме написали кода на неправилно място.

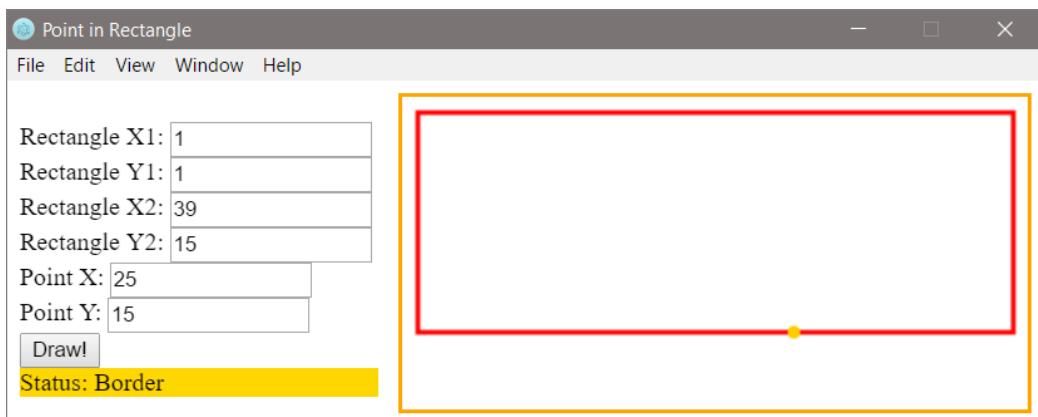
Накрая стартираме приложението в собствен GUI прозорец:

```
electron .
```

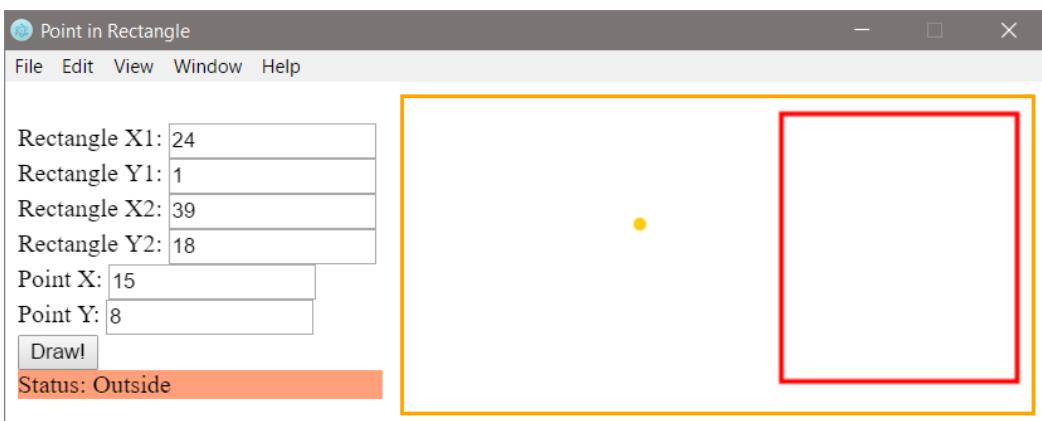
Случай 1: Точката се намира в правоъгълника:



Случай 2: Точката лежи на една от страните на правоъгълника:



Случай 3: Точката се намира извън правоъгълника:



Ако имате трудности с последната задача, питайте във форума на СофтУни:  
<https://softuni.bg/forum>.

# Глава 4.2. По-сложни проверки – изпитни задачи

В предходната глава се запознахме с **вложените условни конструкции** в езика **JavaScript**. Чрез тях програмната логика в дадена програма може да бъде представена посредством **if конструкции**, които се съдържат една в друга. Разглеждахме 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;  
}
```

Вложените **if**-конструкции са проверка (**if**) и вътре в нея друга проверка (**if**).



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.
- 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

Програмата чете **четири цели числа** (аргумента):

- Първият ред (аргумент) съдържа **час на изпита** – цяло число от 0 до 23.
- Вторият ред (аргумент) съдържа **минута на изпита** – цяло число от 0 до 59.
- Третият ред (аргумент) съдържа **час на пристигане** – цяло число от 0 до 23.
- Четвъртият ред (аргумент) 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	Late 20 minutes after the start	16 00	Early 1:00 hours before the start	9 00	On time 30 minutes before the start
9 9 50		15 00		8 30	

Вход	Изход	Вход	Изход	Вход	Изход
9 00	Late 1:30 hours after the start	14 00	On time 5 minutes before the start	10 00	
10 30		13 55		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** параметъра с различни **цели числа**. Разглеждайки дадените параметри можем да се спрем на типът **Number**, тъй като той удовлетворява очакваните ни стойности. Четем входните параметри и **парсваме** текстовите стойности към избрания от нас тип данни за **цяло число**.

```
let examHours = Number(arg1);
let examMinutes = Number(arg2);
let arrivalHours = Number(arg3);
let arrivalMinutes = Number(arg4);
```

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.

```
let late = "Late";
let onTime = "On time";
let 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:

```
let examTime = (examHours * 60) + examMinutes;
```

Нека изчислим по същата логика и времето на пристигане на студента:

```
let arrivalTime =
    (arrivalHours * 60) + arrivalMinutes;
```

Остава ни да пресметнем разликата в двете времена, за да можем да определим кога и в **какво време спрямо изпита** е пристигнал студентът:

```
let 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** ( $\leq 0$ ), 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:

```
let studentArrival = late;
if (totalMinutesDifference < -30) {
    studentArrival = early;
} else if (totalMinutesDifference <= 0) {
    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.

```
let result = "";
if (totalMinutesDifference != 0) {
    let hoursDifference =
        Math.abs(Math.floor(totalMinutesDifference / 60));
    let minutesDifference =
        Math.abs(totalMinutesDifference % 60);

    if (hoursDifference > 0) {
        result =
            hoursDifference + ":" +
            (minutesDifference > 9
                ? minutesDifference
                : "0" + minutesDifference) +
            " hours";
    } 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.log(studentArrival);
if (result) {
    console.log(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 тряба 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/932#0>.

## Задача: пътешествие

Странно, но повечето хора си планират 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, който програмата чете се състои от **два реда** (аргумента):

- На първия ред (аргумент) получаваме **бюджета** - реално число в интервал [10.00 ... 5000.00].
- На втория ред (аргумент) – **един** от двата възможни сезона: "summer" или "winter".

## Изходни данни

На конзолата трябва да се отпечатат **два** реда.

- На първи ред – "Somewhere in {дестинация}" измежду "Bulgaria", "Balkans" и "Europe".
- На втори ред – "{Вид почивка} – {Похарчена сума}".
  - Почивката 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** параметъра 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 **Number** като тип за бюджета, а за сезона – **String**.

```
let budget = Number(arg1);
let season = arg2.toLowerCase();
```



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!

## Изчисления

Нека си създадем и инициализираме нужните за логиката и изчисленията променливи:

```
let destinationResult = "";
let holidayInformation = "";
let moneySpent = 0.00;
```

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.

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!**

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 <= 100.00) {
    destinationResult = "Bulgaria";
    if (season === "summer") {
        moneySpent = 0.30 * budget;
        holidayInformation =
            "Camp - " + moneySpent.toFixed(2);
    } else {
        moneySpent = 0.70 * budget;
```

```

        holidayInformation =
            "Hotel - " + moneySpent.toFixed(2);
    }
} else if (budget <= 1000.00) {
    destinationResult = "Balkans";
    if (season === "summer") {
        moneySpent = 0.40 * budget;
        holidayInformation =
            "Camp - " + moneySpent.toFixed(2);
    } else {
        moneySpent = 0.80 * budget;
        holidayInformation =
            "Hotel - " + moneySpent.toFixed(2);
    }
} else {
    destinationResult = "Europe";
    moneySpent = 0.90 * budget;
    holidayInformation =
        "Hotel - " + moneySpent.toFixed(2);
}

```

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 <= 100.00) {
    destinationResult = "Bulgaria";
    if (season === "summer") {
        moneySpent = 0.30 * budget;
        holidayInformation =
            "Camp - " + moneySpent.toFixed(2);
    } else {
        moneySpent = 0.70 * budget;
        holidayInformation =
            "Hotel - " + moneySpent.toFixed(2);
    }
}

```

може да бъде съкратен до този вид:

```

destinationResult = "Bulgaria";
moneySpent = 0.70 * budget;
holidayInformation =
    "Hotel - " + moneySpent.toFixed(2);

```

```

if (season === "summer") {
    moneySpent = 0.30 * budget;
    holidayInformation =
        "Camp - " + moneySpent.toFixed(2);
}

```

## Отпечатване на резултата

Остава ни да покажем изчисления резултат на конзолата:

```

console.log("Somewhere in " + destinationResult);
console.log(holidayInformation);

```

## Тестване в Judge системата

Тествайте решението си: <https://judge.softuni.bg/Contests/Practice/Index/932#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

На функцията се подават **3 аргумента**:

- 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**:

- " $\{N1\} \% \{N2\} = \{\text{остатък}\}$ ".
- 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
123	
12	$123 / 12 = 10.25$
/	

Вход	Изход
10	
3	$10 \% 3 = 1$
%	

Вход	Изход
10	
12	$10 + 12 = 22$ – even
+	

Вход	Изход
112	
0	Cannot divide 112 by zero
/	

## 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** параметъра с входни данни. На първите **два** параметъра ни се подават **цели числа** (в указания от заданието диапазон), а на третия - **аритметичен символ**.

```
let N1 = Number(arg1);
let N2 = Number(arg2);
let nOperator = arg3;
```

### Изчисления

Нека си създадем и инициализираме нужните за логиката и изчисленията променливи. В едната ще пазим **резултата от изчисленията**, а другата ще използваме за **краиния изход** на програмата.

```
let result = 0.00;
let output = "";
```

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 дали операцията е **деление** или **модулно деление**, след което да инициализираме резултата.

```
if (N2 === 0 && (nOperator === "/" || nOperator === "%")) {
    output = "Cannot divide " + N1 + " by zero";
}
```

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 (+)**, **изважддане (-)** 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 === "/") {
    result = (N1 / N2).toFixed(2);
    output = N1 + " " + nOperator + " " +
    N2 + " = " + result;
} else if (nOperator === "%") {
    result = N1 % N2;
    output = N1 + " " + nOperator + " " +
    N2 + " = " + result;
}
```

Завършваме с проверките за събиране, изважддане и умножение:

```
else {
    if (nOperator === "+") {
        result = N1 + N2;
    } else if (nOperator === "-") {
        result = N1 - N2;
    } else if (nOperator === "*") {
        result = N1 * N2;
    }
```

```

    output =
        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 с и без тернарен оператор.

Без използване на тернарен оператор кодът е по-дълъг, но се чете лесно:

```

let numberIs = "";
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:

```
let numberIs = result % 2 == 0 ? "even" : "odd";
```

## Отпечатване на резултата

Накрая ни остава да покажем изчисления резултат на конзолата:

```
console.log(output);
```

## Тестване в Judge системата

Тествайте решението си: <https://judge.softuni.bg/Contests/Practice/Index/932#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

Програмата прочита **точно 3 реда** (аргумента):

- На **първия** ред (аргумент) е **бюджетът** – реално число в интервала [1 000.00 ... 1 000 000.00].
- На **втория** ред (аргумент) е **категорията** – "VIP" или "Normal".
- На **третия** ред (аргумент) 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 $499.99 * 49 = 24495.51$ .

Вход	Изход	Обяснения
		$24499.510000000002 < 18000.$ <i>Не стигат</i> $24499.51 - 18000 = *6499.51$

## 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:

```
let budget = Number(arg1);
let ticketType = arg2;
let people = Number(arg3);
```

### Изчисления

Нека създадем и инициализираме нужните за изчисленията променливи:

```
let transportCharges = 0.00;
let moneyForTickets = 0.00;
let moneyDifference = 0.00;
```

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 = 0.75 * budget;
} else if (people <= 9) {
    transportCharges = 0.60 * budget;
```

```

} else if (people <= 24) {
    transportCharges = 0.50 * budget;
} else if (people <= 49) {
    transportCharges = 0.40 * budget;
} else if (people >= 50) {
    transportCharges = 0.25 * 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 * 249.99;
        break;
    case "VIP":
        moneyForTickets = people * 499.99;
        break;
    default:
        moneyForTickets = people * 249.99;
        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;
let result =
    "Yes! You have " + moneyDifference.toFixed(2) + " leva left.";

if (moneyDifference < 0) {
    result =
        "Not enough money! You need " +
        Math.abs(moneyDifference).toFixed(2) + " leva.";
}

```

## 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/932#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

Програмата прочита **точно два реда** (аргумента):

- На първия ред (аргумент) е **месецът** – **May, June, July, August, September** или **October**.
- На втория ред (аргумент) 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 за целия престой да е форматирана с точност до **два символа** след десетичния знак.

## Примерен вход и изход

Вход	Изход	Вход	Изход
June 14	Apartment: 961.80 lv. Studio: 1052.80 lv.	August 20	Apartment: 1386.00 lv. Studio: 1520.00 lv.

Вход	Изход	Обяснения
May 15	Apartment: 877.50 lv. Studio: 525.00 lv.	През <b>май</b> , при повече от <b>14 нощувки</b> , намаляваме цената на <b>студиото</b> с <b>30%</b> ( $50 - 15 = 35$ ), а на <b>апартамента</b> – с <b>10%</b> ( $65 - 6.5 = 58.5$ ). Целият престой в <b>апартамент</b> – <b>877.50 лв.</b> . Целият престой в <b>студио</b> – <b>525.00 лв.</b> .

## 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 параметъра, съдържащи входните данни - първият параметър е **месецът**, през който се планува **престой**, а вторият - **броят нощувки**.

Нека обработим и запазим входните данни в подходящи за това параметри:

```
let month = arg1;
let nights = Number(arg2);
```

## Изчисления

След това да създадем и инициализираме нужните за изчисленията променливи:

```
let studioPrice = 50.00;
let apartmentPrice = 65.00;
let studioRent = 0.00;
let apartmentRent = 0.00;
```

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 настанияване - 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 = 50.00;
        apartmentPrice = 65.00;

        studioRent = studioPrice * nights;
        apartmentRent = apartmentPrice * nights;

        if (nights > 14) {
            studioRent *= 0.70;
            apartmentRent *= 0.90;
        } else if (nights > 7) {
            studioRent *= 0.95;
        }
        break;
}
```

За следващите месеци логиката и изчисленията ще са донякъде **идентични**:

```
case "June":
case "September":
    studioPrice = 75.20;
    apartmentPrice = 68.70;

    studioRent = studioPrice * nights;
    apartmentRent = apartmentPrice * nights;

    if (nights > 14) {
        studioRent *= 0.80;
        apartmentRent *= 0.90;
    }
    break;
```

```

case "July":
case "August":
    studioPrice = 76.00;
    apartmentPrice = 77.00;

    studioRent = studioPrice * nights;
    apartmentRent = apartmentPrice * nights;

    if (nights > 14) {
        apartmentRent *= 0.90;
    }
    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 запищем в изходните ни параметри - **studioInfo** и **apartmentInfo**:

```

let studioInfo = "Studio: " +
                 studioRent.toFixed(2) + " lv.";
let apartmentInfo = "Apartment: " +
                     apartmentRent.toFixed(2) + " lv.";

```

За изчисленията на изходните параметри използваме метода **.toFixed(Number)**. Този метод закръгля десетично число до зададен брой цифри след десетичния знак. За целта, използваме метода, за да закръглим двете ни променливи (**studioRent**, **apartamentPrice**). 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/932#4>.

# 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, изискващи повторение на поредица от действия, като използваме цикли.

## Видео

Гледайте видео-урок по тази глава тук: <https://youtu.be/jlcW93jb-8>.

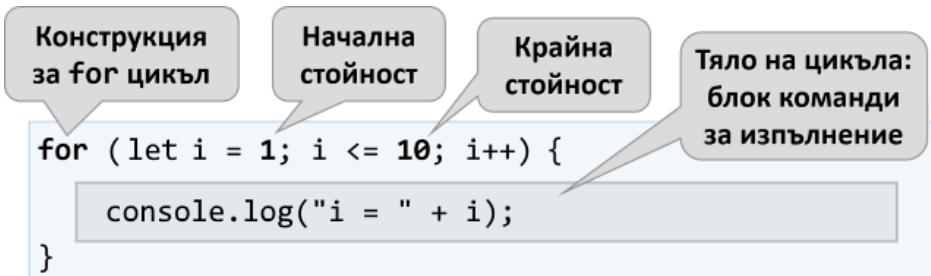
### Повторения на блокове код (for цикъл)

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 последователно през числата от 1 до 10 и ги отпечатва:

```
for (let i = 1; i <= 10; i++) {  
    console.log("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 **крайна стойност**. Тялото на цикъла обикновено се огражда с къдрави скоби **{ }** и представлява блок с една или няколко команди. На фигурата 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 и 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 помощта на променливата **i**, през числата от 1 до 100 и ги печатаме в тялото на цикъла:

```
function numbers1To100() {
    for (let i = 1; i <= 100; i++) {
        console.log(i);
    }
}
```

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

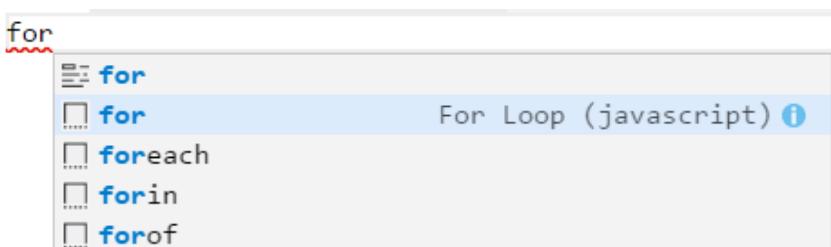
PROBLEMS	OUTPUT	DEBUG CONSOLE	TERMINAL
		89 90 91 92 93 94 95 96 97 98 99 100	

## Тестване в Judge системата

Тествайте решението си: <https://judge.softuni.bg/Contests/Practice/Index/933#0>.

## Code Snippet за for цикъл във Visual Studio Code

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 Code**. Напишете **for** в редактора за JavaScript код във Visual Studio Code и натиснете един път [Tab]:



VS Code ще разгъне за вас шаблон и ще напише цялостен **for цикъл**:

```
for (let index = 0; index < array.length; index++) {
    const element = array[index];
}

}
```

Опитайте сами, за да усвоите умението да ползвате шаблона за код за **for цикъл** във Visual Studio Code.

## 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**:

```
function numbersEndingIn7() {
    for (let 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/933#1>.

## Пример: всички латински букви

Да се напише програма, която отпечатва буквите от латинската азбука:

- a, b, c, ..., z

### Насоки и подсказки

Може да решим задачата като завъртим **for цикъл**, който преминава последователно през кодовете на всички букви от латинската азбука, като съобразим, че кодът (поредният номер в [Unicode номерацията](#) на буквите и символите) на 'a' е 97, кодът на 'b' е 98 и т.н., а кодът на 'z' е 122. Преминаването от номер на буква към самата буква става с функцията **String.fromCharCode(x)**. Ето примерна реализация:

```
function latinLetters() {
    for (let i = 97; i <= 122; i++) {
        console.log(String.fromCharCode(i));
    }
}
```

Ако искаме да направим кода с една идея по-четим, можем да го напишем така:

```
function latinLetters() {
    for (let ch = 'a'.charCodeAt(0); ch <= 'z'.charCodeAt(0); ch++) {
        console.log(String.fromCharCode(ch));
    }
}
```

Така става много по-ясно през кои стойности преминава цикълът и няма вълшебни числа като 97 и 122.

## Тестване в Judge системата

Тествайте решението си: <https://judge.softuni.bg/Contests/Practice/Index/933#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 **args.length**. На всяка стъпка от цикъла четем число **args[i]** и го добавяме към сумата **sum**.
- Накрая отпечатваме получената сума **sum**.

Ето и сорс кода на решението:

```
function sumNumbers(args) {
    let sum = 0;
    for (let i = 1; i < args.length; i++) {
        sum += Number(args[i]);
    }
    console.log(sum);
}
```

## Тестване в Judge системата

Тествайте решението си: <https://judge.softuni.bg/Contests/Practice/Index/933#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. Примери:

### Примерен вход и изход

Вход	Изход
2 100 99	100

Вход	Изход
3 -10 20 -30	20

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 **Number.NEGATIVE\_INFINITY**). С помощта на **for** цикъл, чрез който итерираме **n** пъти (**n = args[0]**), 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.

```
function maxNumber(args) {
    let n = Number(args[0]);
    let max = Number.NEGATIVE_INFINITY;

    for (var i = 1; i <= n; i++) {
        let num = Number(args[i]);

        if (num > max) {
            max = num;
        }
    }

    console.log("max = " + max);
}
```

## Тестване в Judge системата

Тествайте решението си: <https://judge.softuni.bg/Contests/Practice/Index/933#4>.

## Пример: най-малко число

Да се напише програма, която въвежда **n** цели числа (**n > 0**) и намира най-малкото измежду тях. Първо се въвежда броя числа **n**, 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 започваме с друга неутрална начална стойност.

```

function minNumber(args) {
    let n = Number(args[0]);
    let min = Number.POSITIVE_INFINITY;

    for (var i = 1; i <= n; i++) {
        let num = Number(args[i]);

        if (num < min) {
            min = num;
        }
    }

    console.log(min);
}

```

## Тестване в Judge системата

Тествайте решението си: <https://judge.softuni.bg/Contests/Practice/Index/933#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 се печата "Yes" + сумата, иначе 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 намираме 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:

```

function leftRightSum(args) {
    let n = Number(args[0]);
    let leftSum = 0;
    let rightSum = 0;
    for (var i = 1; i <= n; i++) {
        leftSum += Number(args[i]);
    }
    //TODO: make "for" loop and calculate the rightSum

    if (leftSum == rightSum) {
        console.log("Yes, sum = " + leftSum);
    }
    else {
        console.log("No, diff = " + Math.abs(leftSum - rightSum));
    }
}

```

## Тестване в Judge системата

Тествайте решението си: <https://judge.softuni.bg/Contests/Practice/Index/933#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 Sum = 70	5	No	8	No
60		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).

```

function oddEvenSum(args) {
    let n = Number(args[0]);
    let oddSum = 0;
    let evenSum = 0;

    for (var i = 1; i <= n; i++) {
        if (i % 2 == 0) {
            evenSum += Number(args[i]);
        }
        else {
            oddSum += Number(args[i]);
        }
    }
    // TODO: print the sum / difference
}

```

## Тестване в Judge системата

Тествайте решението си: <https://judge.softuni.bg/Contests/Practice/Index/933#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 по-долу:

а	е	и	о	у
1	2	3	4	5

## Примерен вход и изход

Вход	Изход
hello	6 (e+o = 2+4 = 6)
hi	3 (i = 3)

Вход	Изход
bamboo	9 (a+o+o = 1+4+4 = 9)
beer	4 (e+e = 2+2 = 4)

## Насоки и подсказки

Прочитаме входния текст **arg1**, зануляваме сумата и завъртаме цикъл от 0 до **input.length** (дължината на текста). Проверяваме всяка буква **input[i]** дали е гласна и съответно добавяме към сумата стойността ѝ.

```

function vowelSum([arg1]){
    let input = arg1;
    let sum = 0;

    for (var i = 0; i < input.length; i++) {
        if(input[i] == "a"){
            sum += 1;
        }
        else if (input[i] == "e"){
            sum += 2;
        }
        else if (input[i] == "i"){
            sum += 3;
        }
        else if (input[i] == "o"){
            sum += 4;
        }
        else if (input[i] == "u"){
            sum += 5;
        }
    }
    console.log(sum);
}

```

## Тестване в Judge системата

Тествайте решението си: <https://judge.softuni.bg/Contests/Practice/Index/933#8>.

## Какво научихме от тази глава?

Можем да повтаряме блок код с **for** цикъл:

```

for (let i = 1; i <= 10; i++) {
    console.log(i);
}

```

Можем да извършваме различни математически операции:

```

function maxNumber(args) {
    let n = Number(args[0]);
    let max = Number.NEGATIVE_INFINITY;

```

```

for (var i = 1; i <= n; i++) {
    let num = Number(args[i]);
    if (num > max) {
        max = num;
    }
}
console.log("max = " + max);
}

```

## 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:

### Задача: елемент, равен на сумата на останалите

Да се напише програма, която въвежда  $n$  цели числа и проверява дали сред тях съществува число, което е 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
3 1 1 1	No Diff = 1

Вход	Изход
3 5 5 1	No Diff = 1

Вход	Изход
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/933#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 / максимален елемент**, отпечатайте "No".

### Примерен вход и изход

Вход	Изход	Вход	Изход	Вход	Изход
6 2 3 5 4 2 1	OddSum=9, OddMin=2, OddMax=5, EvenSum=8, EvenMin=1, EvenMax=4	2 1.5 -2.5	OddSum=1.5, OddMin=1.5, OddMax=1.5, EvenSum=-2.5, EvenMin=-2.5, EvenMax=-2.5	1	OddSum=1, OddMin=1, OddMax=1, EvenSum=0, EvenMin>No, EvenMax>No

Вход	Изход	Вход	Изход	Вход	Изход
3 -1 -2 -3	OddSum=-4, OddMin=-3, OddMax=-1, EvenSum=-2, EvenMin=-2, EvenMax=-2	1 -5	OddSum=-5, OddMin=-5, OddMax=-5, EvenSum=0, EvenMin>No, EvenMax>No	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**. Припомните 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 получим накрая неутралната стойност, печатаме "No".

## Тестване в Judge системата

Тествайте решението си: <https://judge.softuni.bg/Contests/Practice/Index/933#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
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	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

#### 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/933#11>.

## Упражнения: графични и уеб приложения

В настоящата глава се запознахме с **циклите** 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 фигурки, които се състоят от голям 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!

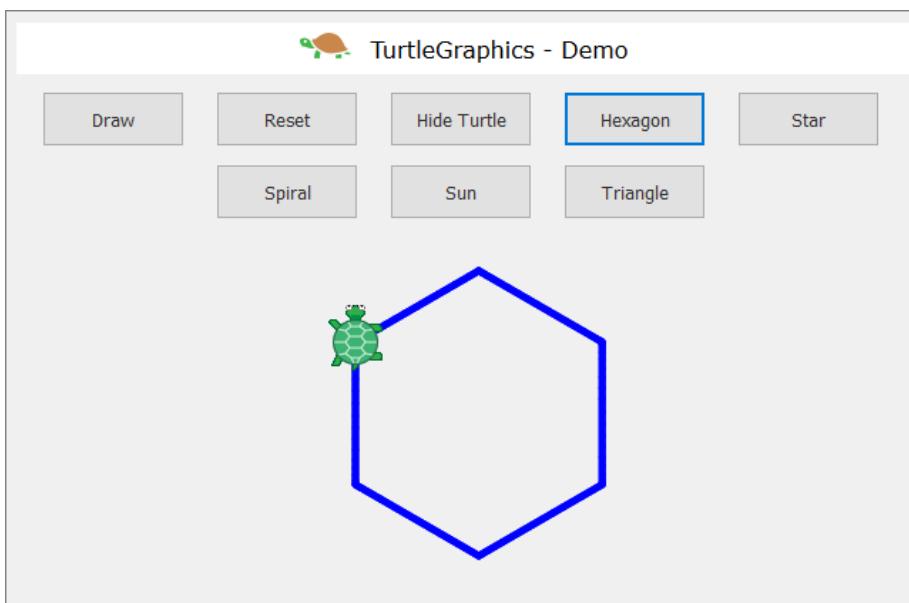
### Задача: чертане с костенурка – графично приложение (GUI)

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".

Нека първо се запознаем с **концепцията за рисуване "Turtle Graphics"**. Може да разгледаме следните източници:

- Дефиниция на понятието "turtle graphics" (концепцията за движение и ротация): <http://c2.com/cgi/wiki?TurtleGraphics>
- Статия за "turtle graphics" в Wikipedia (с повече примери за интересни графики) – [https://en.wikipedia.org/wiki/Turtle\\_graphics](https://en.wikipedia.org/wiki/Turtle_graphics)
- Интерактивен онлайн инструмент за чертане с костенурка – <https://blockly-games.appspot.com/turtle>

Приложението ще изглежда приблизително така:



Ще реализираме приложението като използваме следните технологии:

- Език **HTML** – за описание на потребителския интерфейс (поле за рисуване и бутони).
- **JavaScript** код – за да имплементираме действията на бутоците.
- JS библиотека **jQuery** – за улеснение на достъпа до елементите в потребителския интерфейс.
- JS библиотека **jQuery-Turtle** – за имплементация на чертане по екрана с механиката на “turtle graphics”.

Разполагаме с две възможности, за да създадем заредим всички библиотеки и ресурси за нашето уеб приложение **Turtle Graphics**:

- Зареждане на ресурсите чрез **CDN** (Content Delivery Network).

Този вариант е подходящ, когато имаме постоянна интернет връзка. Нужно е да направим стандартен HTML файл (примерно **index.html**) и запишем в него следния код:

```
<!DOCTYPE html>
<html>
<head>
    <meta charset="UTF-8">
    <title>TurtleGraphics - Demo</title>
    <link rel="stylesheet" type="text/css"
        href="https://cdn.rawgit.com/SoftUni/Programming-Basics-Book-JS-BG/3967382
        8/assets/chapter-5-1-assets/style.css">
</head>

<body>
    <div id="main-frame">
        <div id="header">
            
            <span class="title-box">TurtleGraphics - Demo</span>
        </div>
        <div id="elements">
            <button id="justDraw">Draw</button>
            <button id="reset">Reset</button>
            <button id="hide">Hide Turtle</button>
            <button id="drawHexagon">Hexagon</button>
            <button id="drawStar">Star</button>
            <button id="drawSpiral">Spiral</button>
            <button id="drawSun">Sun</button>
            <button id="drawTriangle">Triangle</button>
            <div id="turtle"></div>
        </div>
    </div>
</body>
```

```

<script
src="https://cdn.rawgit.com/SoftUni/Programming-Basics-Book-JS-BG/39673828
/assets/chapter-5-1-assets/jquery.js"></script>
<script
src="https://cdn.rawgit.com/SoftUni/Programming-Basics-Book-JS-BG/39673828
/assets/chapter-5-1-assets/jquery-turtle.js"></script>
<script> </script>
</body>
</html>

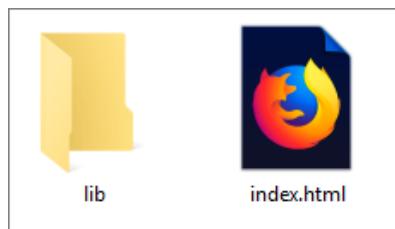
```

Всички необходими ресурси ще се зареждат автоматично при стартирането на файла и можем директно да започнем с въвеждането на нашия **JavaScript** код.

Ако по някаква причина нямате постоянен достъп до Интернет, може да използвате втората опция:

- **Локално зареждане на ресурсите.**

При нея трябва сами да свалите всички необходими файлове и да промените няколко реда в **html** файла. Започнете, като създадете папка с име **Turtle-Demo** и в нея направете основния **html файл** и под-папка за нужните ресурси:



В папка "**lib**" трябва да сложим няколко файла, които можем да свалим от онлайн хранилището на книгата: <https://github.com/SoftUni/Programming-Basics-Book-JS-BG/tree/master/assets/chapter-5-1-assets>.

За ваше улеснение сме сложили файловете и в удобен за сваляне архив **Turtle-Graphics-Demo-Files.zip**:



Нека разгледаме всеки един от тях:

**jquery.js** (версия 2.0.3)

Една от най-известните JavaScript библиотеки, която предлага **бързина и функционалност** при работа с HTML-базиран потребителски интерфейс. Тя

променя начина по който пишем код и планираме неговата структура. Запознайте се с инструмента на адрес: <https://jquery.com>

### jquery-turtle.js (версия 2.0.8)

Плъгин (приставка), написан от **Дейвид Бау** за jQuery - **jQuery-turtle**, който дава набор от функции за **рисуване на графики** от тип "костенурка". Подробна информация и правила за употреба, може да намерите тук: <https://github.com/davidbau/jquery-turtle>.

#### style.css

Набор от **правила за дизайн**, обособени в отделен файл.

#### turtle-icon.png

**Растерна графика**, която използваме с цел по-добра презентация на приложението.

След като сме сложили файловете в папката, трябва да сменим мрежовия адрес на ресурсите в нашия **HTML** файл - **index.html**:

```
<!DOCTYPE html>
<html>
<head>
    <meta charset="UTF-8">
    <title>TurtleGraphics - Demo</title>
    <link rel="stylesheet" type="text/css" href="lib/style.css">
</head>

<body>
    <div id="main-frame">
        <div id="header">
            
            <span class="title-box">TurtleGraphics - Demo</span>
        </div>
        <div id="elements">
            <button id="justDraw">Draw</button>
            <button id="reset">Reset</button>
            <button id="hide">Hide Turtle</button>
            <button id="drawHexagon">Hexagon</button>
            <button id="drawStar">Star</button>
            <button id="drawSpiral">Spiral</button>
            <button id="drawSun">Sun</button>
            <button id="drawTriangle">Triangle</button>
            <div id="turtle"></div>
        </div>
    </div>
</body>
```

```

<script src="lib/jquery.js"></script>
<script src="lib/jquery-turtle.js"></script>
<script> </script>
</body>
</html>

```

След тази промяна, при всяко стартиране на файла, браузъра ни ще зарежда файловете локално от папката "lib".

Сега можем да преминем към забавната част - **писането на JavaScript код** за уеб приложението. Той ще бъде разположен между последната двойка **script** тагове в горния HTML файл (**index.html**):

```
<script> </script>
```

Кодът с функциите на приложението ще бъде сравнително кратък (около 70-80 реда) и затова не е нужно да го отделяме в нов самостоятелен файл. Важно е само да го разположим правилно в нашия HTML файл.



Препоръчително е всички JavaScript файлове да се поставят в края на HTML документа, преди завършващия "**body**" таг. С това гарантираме по-бързото зареждане на страницата, защото не забавямерендрирането (обработването) на елементите и.

Винаги **първо поставяме файла с библиотеката [jquery.js]**, след това този с кода на приставката **[jquery-turtle.js]**. Едва тогава записваме нашия код, защото той се базира на първите два файла. Ако се опитаме да ги разместим ще получим грешки и приложението ни няма да функционира нормално.

Библиотеката "**jQuery**" ни позволява да манипулираме **HTML** елементи, като използваме валидни селектори за дизайн (**CSS**). Нужно е да приложим конкретен синтаксис:

```
$('#id') или $('.class')
```

Можем да използваме името на **HTML** елемента, **ID** или неговия **клас**. Селекторите винаги са низове от текст, затова се ограждат в единични или двойни кавички. Ако селектора е **ID** (самостоятелно име за всеки елемент), в началото се поставя **знака диез (#)**. Но ако сме решили да селектираме **чрез клас** (едно име за множество елементи), тогава записваме **точка**.

Придържайки се към документацията на **jQuery-turtle**, трябва да инициализираме нашия обект и да зададем основни характеристики на "костенурката". Със следния код ще определим **размера** на графиката (**turtleScale**) и **скоростта** на движение (**turtleSpeed**):

```

eval($.turtle());
$('#turtle').css('turtleScale', '2').css('turtleSpeed', '4');

```

След като сме готови с основата на нашето приложение, остава да напишем функциите за всеки бутон. За тази цел използваме предварително зададените от нас селектори (**ID**) на обектите в **html файла**. Ще споделим кода за първите три бутона, за да се запознаете с основните принципи:

- Бутон "Draw"

Закачаме функция към елемента с ID (селектор) **justDraw**, която да се активира в момента на кликване върху бутона:

```
$('#justDraw').click(function() {
    cg();
    for (let index = 0; index < 4; index++) {
        $('#turtle').pen('blue', '5')
            .lt(30).fd(150)
            .lt(120).fd(150)
            .lt(120).fd(150);
    }
});
```

Първо **изтриваме графиките** с функцията **cg()** (clear graphics) и **построяваме** елементарен **цикъл**, който да се повтори 4 пъти. Целенасочено използваме ключовата дума **let** за дефиниране на променливата **index**. По този начин гарантираме автономността на променливата за конкретния цикъл и може без проблем да използваме същото наименование отново.

При всяка итерация (повторение) прилагаме конкретни методи за движение с определена стойност (завъртане наляво и движение напред):

```
.lt(30) // завъртане наляво (left) с аргумент 30
.fd(150) // движение напред (forward) с аргумент 150
```

Чрез техниката на приковаването (chaining) спестяваме допълнително писане на код:

```
$('#turtle').pen('blue', '5').lt(30).fd(150).lt(120) ...
```

// което е по-кратка версия на класическия метод:

```
$('#turtle').pen('blue', '5');
$('#turtle').lt(30);
$('#turtle').fd(150);
$('#turtle').lt(120);
$('#turtle').fd(150);
...
...
```

- Бутон "Reset"

Закачаме функция към елемента с ID (селектор) "reset", която да се активира в момента на кликване:

```
$('#reset').click(function() {
    window.location.reload();
});
```

Чрез **window.location.reload()** активираме презареждане на прозореца, с което нулираме текущото му състояние. Важно е да отбележим, че **location** е характеристика на обекта **window**, а **reload()** е метод на **location**.

- Бутон "Hide Turtle"

Закачаме функция към елемента с ID (селектор) "hide", която да се активира при кликване върху бутона:

```
$('#hide').click(function() {
    $('#turtle').toggle();
    $(this).text(function(i, text) {
        return text === "Hide Turtle" ?
            "Show Turtle" : "Hide Turtle";
    });
});
```

Използваме готова функция от библиотеката **jQuery - toggle()**. Чрез нея скриваме и показваме елементи. Отделно ще прикачим друга функция, която да променя текста на бутона при кликване. Трябва да използваме ключовата дума **this**. Тя има съществена роля в синтаксиса на езика JavaScript и **мислено** може да я заменим със **self/itself**. В конкретния случай е равна на елемента **hide**, т.e. **this = #hide**. Обръщаме се към самия елемент и задаваме функция за смяна на текста, която също се активира при всяко направено кликване върху бутона.

Нека да обобщим кода, който написахме до момента:

```
<script>
eval($.turtle());
$('#turtle').css('turtleScale', '2').css('turtleSpeed', '4');

$('#reset').click(function() {
    window.location.reload();
});
$('#justDraw').click(function() {
    cg();
    for (let index = 0; index < 4; index++) {
        $('#turtle').pen('blue', '5')
            .lt(30).fd(150)
            .lt(120).fd(150)
            .lt(120).fd(150);
```

```

    });
});

$('#hide').click(function() {
    $('#turtle').toggle();
    $(this).text(function(i, text) {
        return text === "Hide Turtle" ? "Show Turtle" :
        "Hide Turtle";
    });
});
</script>

```

Остава да решим проблема с автоматичното изтриване на полето за рисуване при натискане на нов бутон. Не желаем фигурите да се чертаят една върху друга или всеки път да използваме бутона **Reset**.

- Функция **resetCanvas()**

```

function resetCanvas() {
    cg();
    home();
    $('#turtle').css('turtleScale', '2').css('turtleSpeed', '4');
}

```

Прилагаме предварително зададените функции в приставката на Дейвид Бау, за да **изтрием всички графични елементи (cg())** и **преместим** костенурката в **стартовата и позиция (home())**. След това задаваме отново първоначалните настройки на елемента **turtle**. Цялата функция **resetCanvas()** се прибавя в началото на всяка нова функция, която ще прикачим към останалите бутони.

Пример:

```

$('#drawSpiral').click(function() {
    resetCanvas();
    $('#turtle').css('turtleSpeed', '4');
    for (let index = 0; index < X; index++) {
        // replace "X" with an appropriate number
        // some code you need to add
    }
});

```

По ваша преценка, можете да променяте скоростта на анимацията и цвета на контура, като добавяте новата настройка директно в функцията **click(...)** и приложената в него функция за всеки бутон:

```

$('#turtle').css('turtleSpeed', '6').pen('red', '5');

```

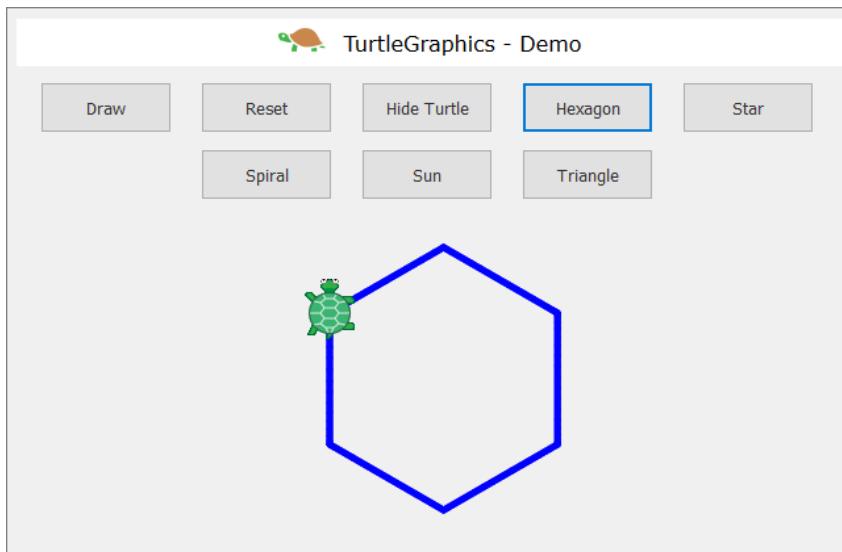
## Задача: \* чертане на шестоъгълник с костенурката

Добавете функция за бутон [Hexagon], който чертае правилен шестоъгълник.

Подсказка:

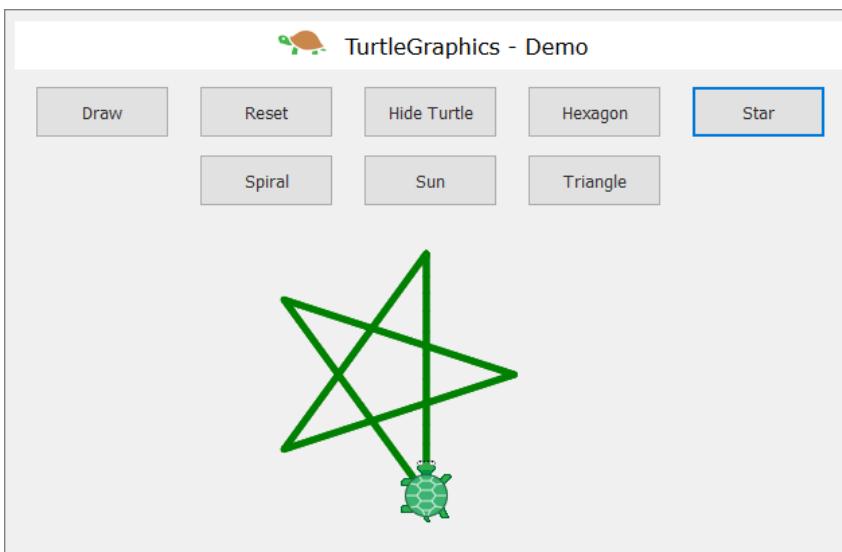
В цикъл повторете 6 пъти следното:

- Ротация на 60 градуса.
- Движение напред с 90 пиксела.



## Задача: \* чертане на звезда с костенурката

Добавете функция за бутон [Star], който чертае звезда с 5 върха (петолъчка), като на фигурата по-долу:



**Подсказка:**

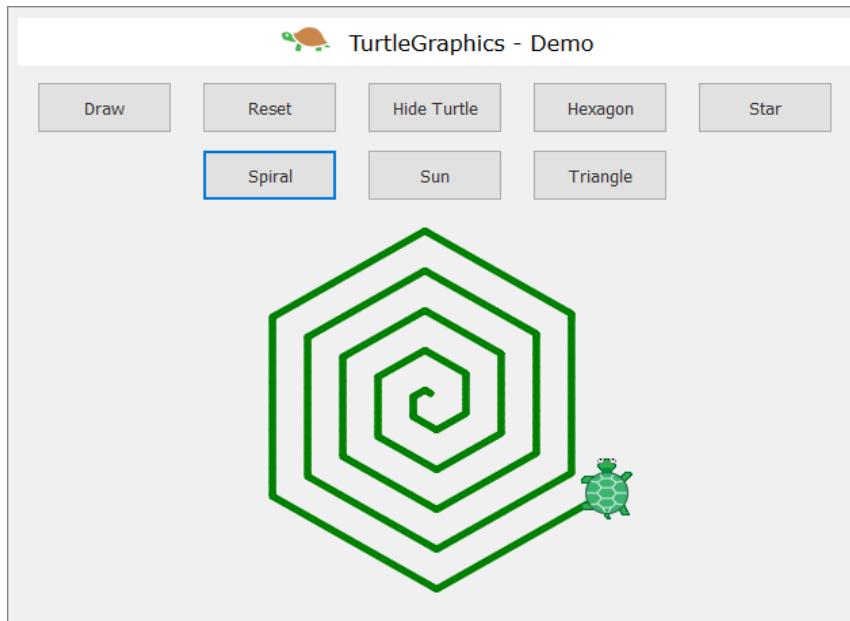
Сменете цвета: `$("#turtle").pen("green", "5")`

В цикъл повторете 5 пъти следното:

- Движение напред със 180.
- Ротация на 144 градуса.

### Задача \* чертане на спирала с костенурката

Добавете функция за бутон [Spiral], който чертае спирала с 30 върха като на фигурата по-долу:



**Подсказка:**

Чертайте в цикъл, като движите напред и завъртате. С всяка стъпка увеличавайте постепенно дължината на движението с 5 напред и завъртайте на 60 градуса.

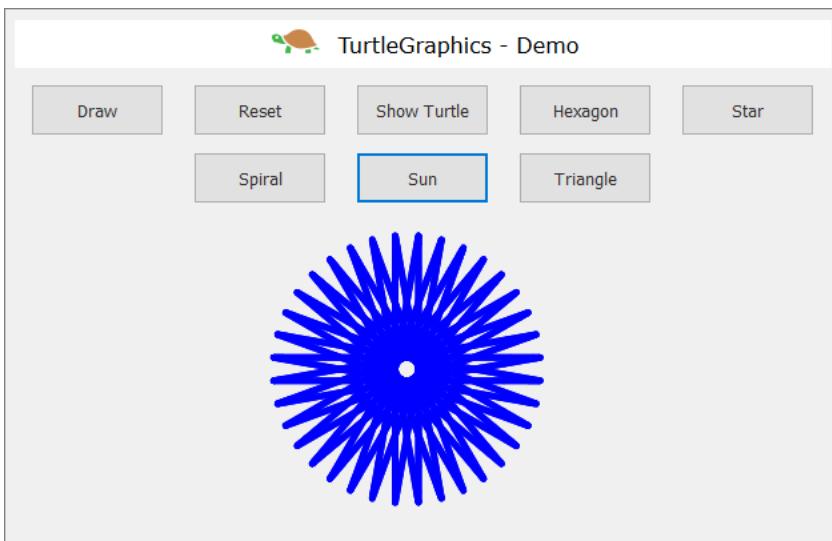
### Задача: \* чертане на слънце с костенурката

Добавете функция за бутон [Sun], който чертае слънце с 36 върха като на фигурата по-долу.

**Подсказка:**

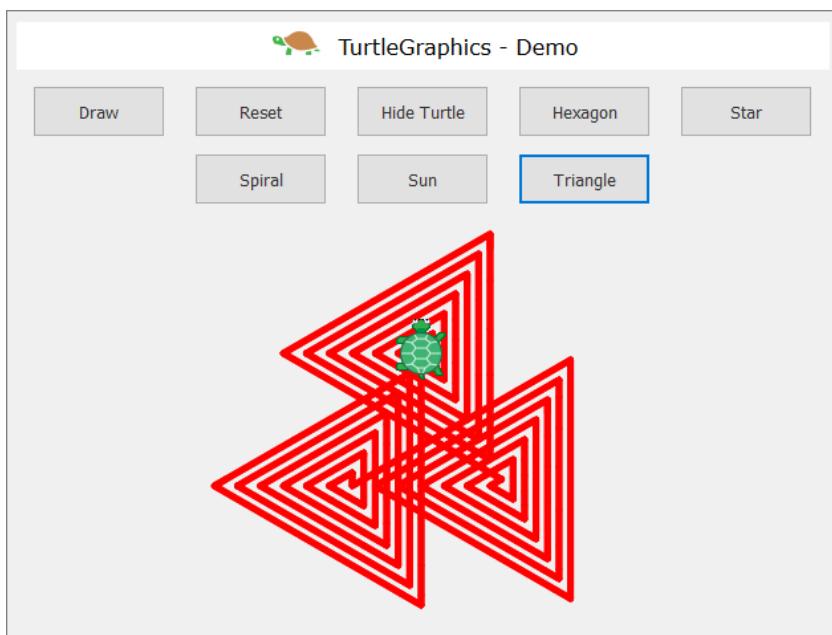
В цикъл повторете 36 пъти следното:

- Движение напред с 200.
- Ротация на 170 градуса.



### Задача: \* чертане на спирален триъгълник с костенурката

Добавете функция за бутон [Triangle], който чертае три триъгълника с по 22 върха като на фигурата по-долу:



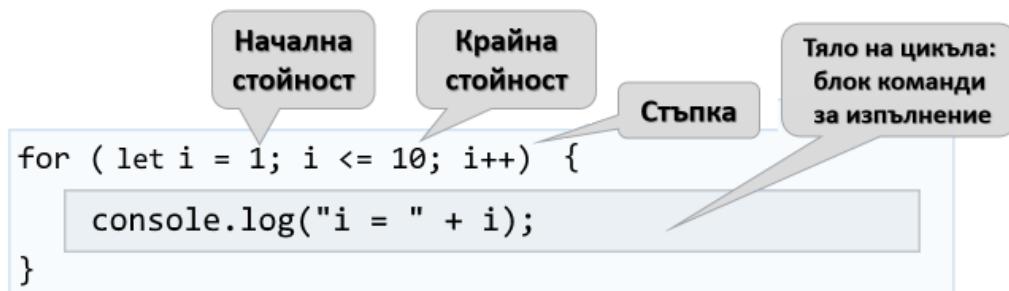
#### Подсказка:

Чертайте в цикъл като движите напред и завъртате. С всяка стъпка увеличавайте с 10 дължината на движението напред и завъртайте на 120 градуса. Повторете в още един цикъл 3 пъти за трите триъгълника.

Ако имате проблеми с примерния проект по-горе, питайте във форума на СофтУни: <https://softuni.bg/forum>.

# 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 разгледахме 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 (**let 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	$p4 = 3 / 20 * 100 = 15.00\%$
$\geq 800$	920, 800	2	$p5 = 2 / 20 * 100 = 10.00\%$

## Входни данни

На първия ред (аргумент) 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 (аргумента) стои по едно цяло число в интервала [1 ... 1000] – числата, върху които да бъде изчислена хистограмата.

## Изходни данни

Да се отпечатат 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
3	66.67%	4	75.00%
1	0.00%	53	0.00%
2	0.00%	7	0.00%
999	0.00%	56	0.00%
	33.33%	999	25.00%

Вход	Изход	Вход	Изход	Вход	Изход
9	33.33%	14	57.14%	7	14.29%
367	33.33%	53	14.29%	800	28.57%
99	11.11%	7	7.14%	801	14.29%
200	11.11%	56	14.29%	250	14.29%
799	11.11%	180	7.14%	199	28.57%
999		450			

Вход	Изход	Вход	Изход	Вход	Изход
333		920		399	
555		12		599	
111		7		799	
9		150			
		250			
		680			
		2			
		600			
		200			

## Насоки и подсказки

Програмата, която решава този проблем, можем да разделим мислено на три части:

- **Прочитане на входните данни** – в настоящата задача това включва прочитането на числото `n`, последвано от `n` на брой цели числа, всяко на отделен ред.
- **Обработка на входните данни** – в случая това означава разпределение на числата по групи и изчисляване на процентното разделение по групи.
- **Извеждане на краен резултат** – отпечатване на хистограмата на конзолата в посочения формат.

### Прочитане на входните данни

Преди да преминем към самото прочитане на входните данни трябва да декларираме променливите, в които ще ги съхраняваме:

```
//Променлива, която съдържа общия брой на числата
let n = Number(args[0]);

//Променливи, пазещи броя числа по групи
let p1 = 0;
let p2 = 0;
let p3 = 0;
let p4 = 0;
let p5 = 0;

//Променливи, в които ще запазим
//процентното разделение на отделните групи
let p1Percentage = 0;
let p2Percentage = 0;
```

```
let p3Percentage = 0;
let p4Percentage = 0;
let p5Percentage = 0;
```

В променливата **n** ще съхраняваме броя на числата, които ще трябва да четем. Допълнително си декларираме и променливите **p1**, **p2** и т.н., в които ще пазим броя на числата от съответната група.

След като сме си декларирали нужните променливи, можем да пристъпим към обработката на входните данни.

## 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). Всяка итерация на цикъла ще прочита и разпределя **едно единствено** число (**currentNum**) в съответната му група. За да определим дали едно число принадлежи към дадена група, **правим проверка в съответния ѝ диапазон**. Ако това е така - увеличаваме броя на числата в тази група (**p1**, **p2** и т.н.) с 1:

```
for (let i = 1; i <= n; i++) {
    let currentNum = Number(args[i]);
    if (currentNum < 200) {
        p1++;
    } else if (currentNum < 400) {
        p2++;
    } else if (currentNum < 600) {
        p3++;
    } else if (currentNum < 800) {
        p4++;
    } else {
        p5++;
    }
}
```

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:

$$(\text{процент на група}) = (\text{брой числа в група}) / (\text{брой на всички числа}) * 100$$

Тази формула в програмния код изглежда по подобие на кода по-долу::

```
p1Percentage = (p1 / n * 100).toFixed(2);
```

В условието е казано, че процентите трябва да са **с точност две цифри след десетичната точка**. Имайки това предвид, към формулата добавяме метода **.toFixed(...)** и за първата променлива тя ще изглежда така:

```
p1Percentage = (p1 / n * 100).toFixed(2);
//Добавете формулите 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:

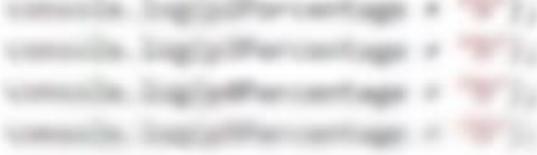
- **i = 0** – прочитаме числото 1, което е по-малко от 200 и попада в първата група (**p1**), увеличаваме брояча на групата с 1.
- **i = 1** – прочитаме числото 2, което отново попада в първата група (**p1**) и увеличаваме брояча ѝ отново с 1.
- **i = 2** – прочитаме числото 999, което попада в последната група (**p5**), защото е по-голямо от 800, и увеличаваме брояча на групата 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.

### Извеждане на краен резултат

Остава само да отпечатаме получените резултати:

```
console.log(p1Percentage + "%");
```



### Тестване в Judge системата

Тествайте решението си: <https://judge.softuni.bg/Contests/Practice/Index/934#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 ( $2 \rightarrow 10, 4 \rightarrow 20, 6 \rightarrow 30$  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 купи пералня**.

## Входни данни

Програмата прочита **3 числа** (аргумента), въведени от потребителя, на отделни редове:

- Възрастта на Лили – **цяло число** в интервала **[1 ... 77]**.
- Цената на **пералнята** – **число** в интервала **[1.00 ... 10 000.00]**.
- Единична цена на **игралка** – **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	<p>Първи рожден ден получава <b>игралка</b>; <b>2ри</b> <math>\rightarrow</math> <b>10 лв.</b>;  <b>3ти</b> <math>\rightarrow</math> <b>игралка</b>; <b>4ти</b> <math>\rightarrow</math> <math>10 + 10 = 20</math> лв.; <b>5ти</b> <math>\rightarrow</math> <b>игралка</b>;  <b>6ти</b> <math>\rightarrow</math> <math>20 + 10 = 30</math> лв.; <b>7ми</b> <math>\rightarrow</math> <b>игралка</b>; <b>8ми</b> <math>\rightarrow</math> <math>30 + 10 = 40</math> лв.; <b>9ти</b> <math>\rightarrow</math> <b>игралка</b>; <b>10ти</b> <math>\rightarrow</math> <math>40 + 10 = 50</math> лв.  <b>Спестила е</b> <math>\rightarrow</math> <math>10 + 20 + 30 + 40 + 50 = 150</math> лв..  <b>Продала е</b> 5 <b>игралки</b> по <b>6 лв.</b> = <b>30 лв..</b>  <b>Брат ѝ взел</b> 5 <b>пъти</b> по <b>1 лев</b> = <b>5 лв.</b> <b>Остават</b> <math>\rightarrow</math> <math>150 + 30 - 5 = 175</math> лв. <b>175 &gt;= 170</b>(цената на <b>пералнята</b>)  <b>успяла е</b> да я <b>купи</b> и са ѝ <b>останали</b> <math>175 - 170 = 5</math> лв.</p>

Вход	Изход	Коментари
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.

```
let age = Number(arg1);
let washingMachinePrice =
let toyPrice =
```

Отново започваме с избора на подходящи имена на променливите. За годините на Лили (**age**), цената на пералнята (**washingMachinePrice**) и за единичната цена на играчката (**toyPrice**). В кода по-горе декларираме и инициализираме (присвояваме стойност) също и променливите за броя на играчките (**toysCount**), както и парите от рожденияте дни (**moneyFromBirthdays**):

```
let toysCount = 0;
let moneyFromBirthdays = 0;
```

С **for цикъл** преминаваме през всеки рожден ден на Лили. 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**. Обратно, когато водещата променлива е **четно число**, това означава, че Лили е **получила пари** и съответно прибавяме тези пари към общите ѝ спестявания. **Увеличаваме** стойността на променливата **moneyFromBirthdays**, т.е. **увеличаваме с 10** сумата, която тя ще получи на следващия си рожден ден. Едновременно с това **изваждаме по 1 лев** – парите, които брат ѝ взема. Използваме съкратено записване, като добавяме два минуса след последния знак на променливата (**moneyFromBirthdays--**):

```
for (let i = 1; i <= age; i++) {
    if (i % 2 == 1){
        toysCount++;
    } else {
        moneyFromBirthdays += 10 * i/2;
        moneyFromBirthdays--;
    }
}
```

Вероятно ще се затрудните с пресмятането на парите за рождените дни, ако оставите бонуса да се натрупва по следния начин:

```
moneyFromBirthdays += 10;
```

Финалният резултат е  $10 \times 5 = 50$ , докато на нас ни е нужен  $10 + 20 + 30 + 40 + 50 = 150$ . Можем да решим проблема с допълнителна променлива (**bonusMoney**):

```
bonusMoney += 10;
moneyFromBirthdays += bonusMoney;
```

Или да включим стойността на променливата **i**, която отброява повторенията и разделим на 2:

```
moneyFromBirthdays += 10 * i/2;
```

Към спестяванията на Лили прибавяме и парите от продадените играчки:

```
let money = moneyFromBirthdays + toyPrice * toysCount;
```

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:

```
if (money >= washingMachinePrice) {
    console.log(`Yes! ${ (money - washingMachinePrice).toFixed(2)} `);
} else {
    console.log(`No! ${ (washingMachinePrice - money).toFixed(2)} `);
}
```

За да спестим създаването на допълнителни променливи, използваме шаблонен литерал - **`\${израз}`**. Той представлява текстов литерал с точно определена поредица от знаци, позволяващ вграждането на изрази. Чрез него можем да извършим изчислението и директно да включим резултата в текстовия низ.

## Тестване в Judge системата

Тествайте решението си: <https://judge.softuni.bg/Contests/Practice/Index/934#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

Програмата прочита **2 числа (аргумента)**, въведени от потребителя на отделни редове:

- 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 **1 ред**. Сумата трябва да е форматирана до **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 $12000 + 19 \cdot 50 = 12950$ → 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 → $-399.85 - 12000 = -12399.85$ <b>12399.85 не достигат</b>

## 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. В условието е казано, че годините на Иванчо са 18, ето защо при декларацията на променливата **years** ѝ задаваме начална стойност **18**. Стойностите на другите променливи прочитаме от подадените параметри на функцията:

```
let heritage = ...;
let yearToLive = ...;
let years = 18;
```

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 \* (годините на Иванчо)**:

```
for (let currentYear = 1800; currentYear <= yearToLive; currentYear++) {
    if (currentYear % 2 == 0) {
        heritage -= 12000;
    } else {
        heritage -= (12000 + 50 * years);
    }
    years++;
}
```

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 метода **Math.abs(...)** при отпечатване на изхода, когато наследството е недостатъчно.

## Тестване в Judge системата

Тествайте решението си: <https://judge.softuni.bg/Contests/Practice/Index/934#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 дена.

Напишете програма, която изчислява за дадения период броя на прегледаните и непрегледаните пациенти.

## Входни данни

На първия ред (аргумент) от входа стои цяло число в интервала [1 ... 1000] – периода, за който трябва да направите изчисления . На следващите редове (аргумента) стои по едно цяло число в интервала [1 ... 10 000] – броя пациенти, които пристигат за преглед за текущия ден.

## 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	Вход	Изход
6	Treated patients: 40. Untreated patients: 87.	3	Treated patients: 21. Untreated patients: 0.
25		7	
25		7	
25		7	
25			
25			
2			

Вход	Изход	Обяснения
4	Treated patients: 23.	1 ден: 7 прегледани и 0 непрегледани пациент за деня
7	Untreated patients: 21.	2 ден: 7 прегледани и 20 непрегледани пациент за деня
27		3 ден: До момента прегледаните пациенти са общо 14, а непрегледаните – 20 –> Назначава се нов лекар –> 8 прегледани и 1 непрегледан пациент за деня
9		4 ден: 1 прегледан и 0 непрегледани пациент за деня
1		Общо: 23 прегледани и 21 непрегледани пациенти.

## Насоки и подсказки

Отново започваме, като **декларираме и инициализираме** нужните променливи. Периодът, за който трябва да направим изчисленията, прочитаме от параметрите на функцията 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:

```
let period = undefined(arg1, arg2);
```

```
let treatedPatients = 0;
let untreatedPatients = 0;
let countOfDoctors = 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 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**).

```

for (let day = 1; day <= period; day++) {
    let currentPatients = Number(args[i]);

    if ((day % 3 == 0) && (untreatedPatients > treatedPatients)) {
        countOfDoctors++;
    }

    if (currentPatients > countOfDoctors) {
        treatedPatients += countOfDoctors;
        untreatedPatients += currentPatients - countOfDoctors;
    } else {
        treatedPatients += currentPatients;
    }
}

```

Накрая трябва само да отпечатаме броя на излекуваните и броя на неизлекуваните пациенти.

## Тестване в Judge системата

Тествайте решението си: <https://judge.softuni.bg/Contests/Practice/Index/934#3>.

## Задача: деление без остатък

Дадени са  $n$  цели числа в интервала  $[1 \dots 1000]$ . 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	$p3 = (5 / 10) * 100 = 50.00\%$

## 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%		
600		9			
200					
800					
799					
199					
46					
128					
65					

## Входни данни

На първия ред (аргумент) 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 25.00%, 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**.

## 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 разглеждахме 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) 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 и да

увеличим стойността на променливата, която пази броя на числата в съответната група.

**Внимание:** **if-else** конструкция в този случай няма да ни свърши работа, защото след като намери съвпадение се прекъсва по-нататъшното проверяване на условията.

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/934#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).
- От 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

Програмата чете поредица от числа (аргумента):

- На първия ред (аргумент): брой на товарите за превоз – цяло число в интервала [1 ... 1000].
- На всеки следващ ред (аргумент) 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 <b>микробус</b> се превозват два от товарите <b>1 + 3</b> , общо <b>4</b> тона.
1	16.00%	С <b>карион</b> се превозва един от товарите: <b>5</b> тона.
5	20.00%	С <b>влак</b> се превозва един от товарите: <b>16</b> тона.
16	64.00%	<b>Сумата</b> от всички товари е: $1 + 5 + 16 + 3 = 25$ тона.
3		Процент товар <b>с микробус</b> : $4/25 * 100 = 16.00\%$ Процент товар <b>с камион</b> : $5/25 * 100 = 20.00\%$ Процент товар <b>с влак</b> : $16/25 * 100 = 64.00\%$ <b>Средна цена</b> на тон превозен товар: $(4 * 200 + 5 * 175 + 16 * 120) / 25 = 143.80$

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 превозват съответно с **микробус**, **карион** 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 подадените аргументи), **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) 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;
```

- При условие, че стойността на **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.

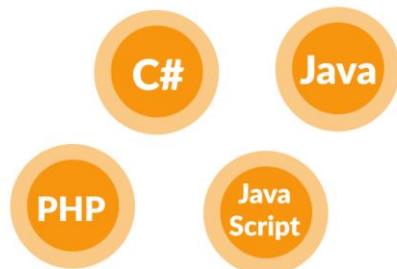
## Тестване в Judge системата

Тествайте решението си: <https://judge.softuni.bg/Contests/Practice/Index/934#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](http://softuni.bg/apply)

# 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 тук: <https://youtu.be/1v1ylZV7p4k>.

### Пример: правоъгълник от 10 x 10 звездички

Да се начертате в конзолата правоъгълник от **10 x 10** звездички.

Вход	Изход
(няма)	***** ***** ***** ***** ***** ***** ***** ***** ***** *****

### Насоки и подсказки

```
function printSquare() {  
    for (let i = 0; i <= 10; i++) {  
        console.log("*".repeat(10));  
    }  
}
```

Как работи примерът? Инициализира се **цикъл** с променлива **i = 0**, която се увеличава на всяка итерация на цикъла, докато е по-малка или равна на **10** (**i <= 10**). Така кодът в тялото на цикъла се изпълнява **10 пъти**. В тялото на цикъла се печата на нов ред в конзолата **"\*".repeat(10)**, което създава низ от 10 звездички.

### Тестване в Judge системата

Тествайте решението си: <https://judge.softuni.bg/Contests/Practice/Index/935#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

```
function drawSquare(n) {
    for (let i = 1; i <= n; i++) {
        console.log("*".repeat(n));
    }
}
```

### Забележка

В някои уеб браузъри еднаквите резултати в конзолата се сливат в един. Препоръчително е да използвате **NodeJS** за примерите. Ако все пак стигнете до този случай, можете да добавите знак за нов ред **\n** на края на метода за отпечатване:  
**console.log("\*".repeat(10) + "\n");**

### Тестване в Judge системата

Тествайте решението си: <https://judge.softuni.bg/Contests/Practice/Index/935#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 **отново**. 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 удовлетворено и започва ново изпълнение на вложени (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  $n$ :

```
function drawSquare(n) {
    for (let i = 1; i <= n; i++) {
        let stars = "";

        for (let j = 1; j <= n; j++) {
            stars += "*";
        }

        console.log(stars);
    }
}
```

Да разгледаме примера по-горе. След инициализацията на **първия (външен)** цикъл, започва да се изпълнява неговото **тяло**, което съдържа **втория (вложена)** цикъл. Той сам по себе запазва низ от  $n$  на брой звездички, в променлива, и след това ги печата на един ред. След като **вътрешният** цикъл **приключи** изпълнението си при първата итерация на външния, то след това **външният ще продължи**. 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 начертат на конзолата квадрат от  $N \times N$  звездички:

Вход	Изход	Вход	Изход	Вход	Изход
2	* *	3	* * * * * * * * *	4	* * * * * * * * * * * * * * * *

## 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:

```
function drawSquare(n) {
    for (let i = 1; i <= n; i++) {
        let stars = "*";

        for (let j = 1; j < n; j++) {
            stars += " *";
        }

        console.log(stars);
    }
}
```

## Тестване в Judge системата

Тествайте решението си: <https://judge.softuni.bg/Contests/Practice/Index/935#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. Нека разгледаме долния пример по-подробно. Виждаме, че **променливата на вложениния цикъл е обвързана с променливата на външния**. По този начин нашата програма печата желания триъгълник:

```
function drawTriangle(n) {
    for (let i = 1; i <= n; i++) {
        let dollars = "$";
```

```

    for (let j = 1; j < i; j++) {
      dollars += " $";
    }

    console.log(dollars);
  }
}

```

## Тестване в Judge системата

Тествайте решението си: <https://judge.softuni.bg/Contests/Practice/Index/935#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:

- 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 отново знак **|**. 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:

```

function drawSquareFrame(n) {
  // print the top row -> + - - +
  let topRow = "+";
  for (let top = 0; top < n - 2; top++) {
    topRow += " -";
  }
}

```

```

topRow += " +";
console.log(topRow);

// print the middle row -> | - - |
for (let mid = 0; mid < n - 2; mid++) {
    let middleRow = "| ";
    for (let j = 0; j < n - 2; j++) {
        middleRow += " -";
    }

    middleRow += " | "
    console.log(middleRow);
}

// print the bottom row -> + - - +
let bottomRow = "+";
for (let bot = 0; bot < n - 2; bot++) {
    bottomRow += " -";
}

bottomRow += " +";
console.log(bottomRow);
}

```

## Тестване в Judge системата

Тествайте решението си: <https://judge.softuni.bg/Contests/Practice/Index/935#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 \*.

Втората (долната) част ще разпечатаме по аналогичен начин, което отново оставяме на читателя да се опита да направи сам:

```
function drawRhombus(n) {
    for (let row = 1; row <= n; row++) {
        let line = "";
        for (let col = 1; col <= n - row; col++) {
            line += " ";
        }

        line += "*";
        for (let col = 1; col < row; col++) {
            line += " *";
        }

        console.log(line);
    }

    // TODO: print the bottom half of the rhombus
}
```

## Тестване в Judge системата

Тествайте решението си: <https://judge.softuni.bg/Contests/Practice/Index/935#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 звездичките и празните места преди и след тях, средната част е **(интервал) | (интервал)**, а последната част са отново звездички, като този път празни места има само **преди** тях. Разпечатването може да бъде постигнато само с **един цикъл** и метода **.repeat(n)**, който ще използваме един път за звездичките и един път за интервалите:

```
function drawTree(n) {
    for (let i = 0; i <= n; i++) {
        let stars = "*".repeat(i);
        let spaces = " ".repeat(n - i);
        let body = " | ";
        let row = spaces + stars + body + stars + spaces;
        console.log(row);
    }
}
```

### Тестване в Judge системата

Тествайте решението си: <https://judge.softuni.bg/Contests/Practice/Index/935#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 x n** като в примерите:

Вход	Изход	Вход	Изход
3	***** * ***** *////*   *////* ***** * *****	4	***** * ***** *////*   *////* *////* *////* ***** * *****

Вход	Изход
5	***** */ * */* *****

## Насоки и подсказки

От примерите виждаме, че очилата могат да се разделят на три части – **горна**, **средна** и **долна**. По-долу е част от кода, с който задачата може да се реши. При рисуването на горния и долнния ред трябва да се изпечатат **2 \* n** звездички, **n** интервала и **2 \* n** звездички:

```
function sunGlasses(n) {
    // Print the top part
    let topLine = "*".repeat(2 * n);
    topLine += " ".repeat(n);
    topLine += "*".repeat(2 * n);
    console.log(topLine);

    for (let i = 0; i < n - 2; i++) {
        // TODO: Print the middle part
    }

    // Print the bottom part
    let bottomLine = "*".repeat(2 * n);
    bottomLine += " ".repeat(n);
    bottomLine += "*".repeat(2 * n);
    console.log(bottomLine);
}
```

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. Проблемът с **(n - 1) / 2 - 1**, е че може да бъде число с десетичен остатък. Пример за **n = 6**:  $(6 - 1) / 2 - 1 \Rightarrow 5 / 2 - 1 \Rightarrow 2.5 - 1 \Rightarrow 1.5$ . Поради тази причина, трябва да приложим математически метод за премахване на десетичната част – **Math.floor(...)**. Методът **Math.floor(...)** връща най-голямото цяло число, което е по-малко или равно на подаденото число:

```
// Print the middle part
for (let i = 0; i < n - 2; i++) {
    let middleLine = "";

    // TODO: print */////*
    if (i === Math.floor((n - 1) / 2 - 1)) {
        middleLine += "|".repeat(n);
    } else {
        middleLine += " ".repeat(n);
    }

    // TODO: print */////*
    console.log(middleLine);
}
```

## Тестване в Judge системата

Тествайте решението си: <https://judge.softuni.bg/Contests/Practice/Index/935#7>.

## Пример: къщичка

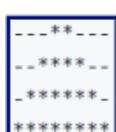
Да се напише програма, която въвежда число  $n$  ( $2 \leq n \leq 100$ ) и печата **къщичка** с размери  $n \times n$ , точно като в примерите:

Вход	Изход	Вход	Изход	Вход	Изход	Вход	Изход
2	** 	3	-*- ***  *	4	- ** - *****   **     **	5	--*-- - *** - *****   ***     ***

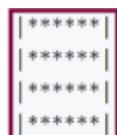
## Насоки и подсказки

Разбираме от условието на задачата, че къщата е с размер  $n \times n$ . Това, което виждаме от примерните вход и изход, е че:

- Къщичката е разделена на 2 части: **покрив** и **основа**.



покрив



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**.

Подаваме **n**, като параметър на нашата функция:

```
function drawHouse(n){
    // ...
}
```



Много е важно да проверяваме дали са валидни входните данни! В тези задачи не е проблем директно да обръщаме подаденият параметър в **Number**, защото изрично е казано, че ще получаваме валидни целичислени 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.

```
let stars = 1;
if (n % 2 === 0){
    stars++;
}
```

Изчисляваме дължината на **покрива**. Тя е равна на половината от **n**. Резултата записваме в променливата **roofLength**:

```
let roofLength = Math.ceil(parseInt(n) / 2);
```

Важно е да се отбележи че, когато **n** е нечетно число, дължината на покрива е по-голяма с един ред от тази на **основата**. В езика **JavaScript**, когато два целочислени типа се делят и има остатък, то резултата ще е десетично число. Пример:

```
let result = 3 / 2; // резултат 1.5
```

Ако искаме да закръглим нагоре, трябва да използваме метода **Math.ceil(...)**: **let result = Math.ceil(3 / 2);**. Резултатът от **3 / 2** е **1.5**. **Math.ceil(...)** ще закръгли резултата от делението нагоре. В нашият случай **1.5** ще се закръгли на **2**. **parseInt()** се използва, за да трансформираме входния параметър в тип **Number**.

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**:

```
let padding = (n - stars) / 2;
```

- Отпечатваме на конзолата: "тиreta" (**padding / 2** на брой пъти) + "звезди" (**stars** пъти) + "тиreta" (**padding / 2** пъти):

```
let line = "-".repeat(padding);
line += "*".repeat(stars);
line += "-".repeat(padding);
console.log(line);
```

- Преди да свърши итерацията на цикъла добавяме 2 към **stars** (броя на звездите):

```
stars += 2;
```

След като сме приключили с **покрива**, е време за **основата**. Тя е по-лесна за печатане:

- Започваме с цикъл от 0 до n (изключено).
- Отпечатваме на конзолата: | + \* (**n - 2** на брой пъти) + |.

```
for (let i = 0; i < n / 2; i++) {
    let line = "|" + "*".repeat(n - 2) + "|";
    console.log(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/935#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  $n \times n$ .

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-долния 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 **тирета**.

### Долна част

- С всеки ред надолу, звездите се събират една с друга. Това означава, че пространството (**тиретата**) между тях намалява, а пространството (**тиретата**) 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 **тирета**, с изключение на средния ред.
- На всеки ред има пространство между двете **звезди**, с изключение на първия и последния ред (понякога **звездата е 1**).

Подаваме стойността на  $n$  като входящ параметър (число) на функция:

```
function drawDiamond(n) {
    // ...
}
```

Започваме да чертаем горната част на диаманта. Първото нещо, което трябва да направим, е да изчислим началната стойност на външната бройка **тиreta leftRight** (тиретата от външната част на **звездите**). Тя е равна на  $(n - 1) / 2$ , закръглено надолу. За закръглянето ще използваме метода **Math.floor(...)**, за да премахнем остатъка от деленето. Такъв може да има при входящо нечетно **n**:

```
let leftRight = Math.floor((n - 1) / 2);
```

След като сме изчислили броя на външните тирета **leftRight**, започваме да чертаем **горната част** на диаманта. Може да започнем, като завъртим **цикъл** от **0** до **n / 2 + 1** (закръглено надолу). Като при всяка итерация на цикъла трябва да се изпълнят следните стъпки:

- Рисуваме по конзолата левите **тирета** (с дължина **leftRight**) и веднага след тях първата **звезда**:

```
console.log("-".repeat(leftRight));
console.log("*");
```

- Ще изчислим разстоянието между двете **звезди**. Може да го изчислим като извадим от **n** дължината на външните **тирета**, както и числото 2 (бройката на **звездите**, т.е. очертанията на диаманта). Резултата от тази разлика записваме в променлива **mid**:

```
let mid = n - 2 * leftRight - 2;
```

- Ако стойността на **mid** е по-малка от 0, то тогава знаем, че на реда трябва да има 1 звезда. Ако е по-голяма или равна 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.log("-".repeat(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, **цикъльт ще е от 0 до  $(n - 1) / 2$** .



**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 **функции**. Можете да потърсите допълнителна информация за тях в Интернет, или да прегледате [глава 10. Функции](#).

Ако сме написали всичко коректно, задачата ни е решена.

## Тестване в Judge системата

Тествайте решението си: <https://judge.softuni.bg/Contests/Practice/Index/935#9>.

## Какво научихме от тази глава?

Запознахме се с конструктора с методът **repeat(...)** на обектите от тип **String**:

```
let foo = "*".repeat(10);
```

Научихме се да чертаем фигури с вложени **for** цикли:

```
for (let i = 1; i <= n; i++) {
    let stars = "";

    for (let j = 1; j <= n; j++) {
        stars += "*";
    }

    console.log(stars);
}
```

## Упражнения: чертане на фигурки в уеб среда

Сега, след като свикнахме с **вложените цикли** и как да ги използваме, за да чертаем фигурки 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 **числов рейтинг** (число от 0 до 100) със звездички. Такава визуализация се среща често в сайтове за електронна търговия, ревюта 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 форум: <https://softuni.bg/forum>.

## Задача: рейтинги – визуализация в уеб среда

Да се разработи **JavaScript** 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.

# Ratings


Отваряме празна папка във файловата система с име "ratings". В нея създаваме два файла и една папка:

- index.html
- script.js
- images (папка)

Сега добавяме **картичките със звездичките** (те са част от файловете със заданието за този проект и могат да бъдат свалени от <https://github.com/SoftUni/Programming-Basics-Book-JS-BG/tree/master/assets/chapter-6-1-assets>). Копираме ги от Windows Explorer и ги поставяме в папката **images** с copy/paste.

Отваряме **index.html** и въвеждаме следният код:

```
<!-- HTML Документ структура -->
<!DOCTYPE html> <!-- HTML5 тип на документа -->
<html lang="en">
    <!-- Заглавна секция, съдържа описателни
        тагове за нашето съдържание -->
    <head>
        <meta charset="UTF-8">
        <meta name="viewport"
            content="width=device-width, initial-scale=1.0">
        <meta http-equiv="X-UA-Compatible" content="ie=edge">
        <!-- Име на документа -->
        <title>Ratings</title>
    </head>
```

```

<!-- Тяло на нашето приложение, описва съдържанието,
което вижда потребителя -->
<body>

    <!-- Заглавие на страницата -->
    <h1>Ratings</h1>

    <!-- Елементи за потребителска интеракция -->
    <input id="input-rating" type="number" name="input-rating"
           min="0" max="100" value="35" />
    <input id="input-draw" type="button"
           name="input-draw" value="Draw" />
    <!-- Нов ред -->
    <br>

    <!-- Елемент, който държи генерираният HTML със звездички -->
    <div id="ratingHolder"></div>

    <!-- Включване на нашето JavaScript приложение -->
    <script src="script.js"></script>
</body>
</html>

```

Този код създава едно поле **input-rating**, в което потребителят може да въвежда число в интервала [0...100] и бутон [Draw], който осъществява пресмятането на звездичките с въведената стойност. Действието, което ще обработи данните, се казва **drawRating**. След формата се отпечатва съдържанието на **<div id="ratingHolder"></div>**. Кодът, който ще се съдържа в него, ще бъде динамично генериран HTML с поредица от звездички.

Добавяме функция **drawRating()** във файла **script.js**, която има следният код:

```

/**
 * drawRating, рисува HTML, който е нужен за визуализацията на
 * звездичките
 * @param {Number} rating
 * @return {String} html
 */
function drawRating(rating) {
    // низ от HTML
    let html = "";
    // краен брой звезди
    let allStars = 10;
    // всички пълни звезди

```

```

let fullStars = Math.floor(rating / allStars);

// всички празни звезди
let emptyStars = Math.floor((100 - rating) / allStars);

// всички наполовина запълнени звезди
let halfStars = allStars - fullStars - emptyStars;

// построяване на HTML
for (let i = 0; i < fullStars; i++) {
    html += '';
}
for (let i = 0; i < halfStars; i++) {
    html += '';
}
for (let i = 0; i < emptyStars; i++) {
    html += '';
}

// връщане на готовият HTML
return html;
}

```

Горният код взима въведеното число **rating**, прави малко пресмятания и изчислява броя **пълни звездички**, броя **половинки звездички** и броя **празни звездички**, след което генерира HTML код, който нареджа няколко картинки със звездички една след друга, за да сглоби от тях картинката с рейтинга. Подгответният HTML код се връща като резултат от функцията и е готов за по-нататъшно използване. Към момента резултатът от тази функция не може да се използва, защото няма как да го свържем с бутона. Въвеждаме функция, която се казва **drawHandler()** и съдържа следният код:

```

/**
 * drawHandler, функция която се изпълнява, когато потребителя
 * клика върху бутона "Draw".
 * @return {Void}
 */
function drawHandler() {
    // намиране на инпут елемента, който държи числото на
    // рейтинга вземане на неговата стойност
    let ratingInput = document.getElementById("input-rating");

    // по подразбиране всички стойности от форми идват като
    // "string" --> обръщаме ги в число чрез "parseInt()"
    let rating = parseInt(ratingInput.value);
}

```

```
// намиране на елемента, който държи звездичките
let ratingHolder = document.getElementById("ratingHolder");

// генериране на HTML на база въведенния рейтинг
let html = drawRating(rating);

// рисуване на страницата
ratingHolder.innerHTML = html;
}
```

Функцията **drawHandler()** прави няколко неща:

- Намира HTML елемента, който държи рейтинга (**input-rating**) и взема неговата стойност.
- Обръща стойността от низ към число.
- Намира HTML елемента, който ще държи звездичките (**ratingHolder**).
- Генерира HTML-а на звездичките, чрез **drawRating(...)** функцията.
- Поставя ново генерираният HTML чрез **innerHTML** метод в елемента **ratingHolder**.

Имаме нужда от още една функция, която да обедини горните две и да ги свърже с HTML елементите. Тази функция се казва **appInit()**, и както името подсказва, нейната роля е да стартира нашето приложение. Въвеждаме следният код във функцията **appInit()**:

```
/**
 * appInit, отговаря за първоначалното изпълнение на нашата програма
 * @return {Void}
 */
function appInit() {
    // намиране на бутон елемента в HTML
    let button = document.getElementById("input-draw");
    // Закачане към събитието "click" за изпълнение на рисуването
    button.addEventListener("click", drawHandler);
    // първоначално изрисуване на рейтинга
    drawHandler();
}
```

След като имаме всички функции въведени е време да стартираме нашето приложение. Обрнете внимание, че **script.js** е включен в края на нашият файл, точно преди затварящия **</body>** таг на страницата. Това е добра практика и осигуря по-бързо зареждане на **DOM** дървото. Това също ни позволява да изпълняваме JavaScript код, който използва HTML елементи. При тези условия можем да бъдем сигурни, че всички те са вече заредени в паметта на браузъра.

Въпреки това, вместо да извикаме директно **appInit()** на края на файла, ще добавим още една добра практика:

```
/**  
 * Стаптиране на приложението асинхронно, чрез "event listener".  
 * Слушане за "DOMContentLoaded".  
 */  
document.addEventListener("DOMContentLoaded", appInit);
```

Събитието **DOMContentLoaded** осигурява, че браузърът е приключил всички действия по създаването на **DOM** дървото. Закачайки се на него с **addEventListener(...)** осигурява правилното изпълнение на JavaScript кода.

Когато браузърът е готов ще изпълни нашата стаптираща функция **appInit()**. Резултатът от функцията е:

- Закачане на функцията **drawHandler()** към събитието **click** върху нашият Draw бутон.
- Първоначално извикване на **drawHandler()**, за да попълним звездичките на база на текущият HTML.

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>.

# 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:

```
let result = "";

for (let i = 0; i < 10; i++) {
    for (let j = 0; j < 10; j++) {
        result += "*";
    }
}

console.log(result);
result = "";
}
```

Запознахме се и с **метода str.repeat(count)**, което дава възможност даден стринг да се печата определен от нас брой пъти:

```
'abc'.repeat(2); // 'abcabc'
```

## 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 приема **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  $n / 2$ .

### Входни данни

Входът на програмата се състои от един елемент (аргумент) - **цяло число n** в интервала [3 ... 1000].

### Изходни данни

Да се отпечатат на конзолата  $n$  текстови реда, изобразяващи **крепостта**, точно както в примерите.

## Примерен вход и изход

Вход	Изход	Вход	Изход	Вход	Изход
3	/^\^/\_\_\\/\_\_\\	4	/^\^\\/\^\\\_\_\\/\_\_\\	5	/^\^\\/_/\^\\\_\_\\/\_\_\\

## Насоки и подсказки

От условието на задачата виждаме, че **входните данни** ще се състоят само от едно **цяло число** в интервала [3 ... 1000], следователно създаваме функция, която приема като аргумент **масив от един елемент**. Тъй като той е от тип **стринг**, а ние трябва да работим с числа, използваме конструктора **Number()** като функция, с която да го конвертираме:

```
function drawFort([arg1]){
    let n = Number(arg1);
```

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 **\**.

По условие лявата и дясната колона във вътрешността си са широки  **$n / 2$** , следователно можем да отделим тази стойност в отделна **променлива**, като внимаваме, че ако като входни данни имаме **нечетно число**, при деление на две резултатът ще е реално число с цяла и дробна част. Тъй като в този случай ни трябва **само цялата част** (в условието на задачата виждаме, че при вход **3** броят на **^** във вътрешността на колоната е **1**, а при вход **5** е **3**), можем да я отделим с метода **Math.trunc()** и да запазим само нейната стойност в новата ни променлива:

```
let colSize = Math.trunc(n/2);
```



Добра практика е винаги, когато видим, че имаме израз, чиято стойност ще използваме **повече от един път**, да запазваме стойността му в променлива. Така от една страна, кодът ни ще

бъде по-лесно четим, от друга страна, ще бъде по-лесно да поправим евентуални грешки в него, тъй като няма да се налага да търсим поотделно всяка употреба на израза.

Декларираме и втора **променлива**, в която ще пазим **стойността** на частта **между двете кули**. Знаем, че по условие общата ширина на крепостта е  **$n * 2$** . Освен това имаме и две кули с по една наклонена черта за начало и край (общо 4 знака) и ширина **colSize**. Следователно, за да получим броя на знаците в междинната част, трябва да извадим размера на кулите от ширината на цялата крепост:  **$2 * n - 2 * colSize - 4$** .

```
let midSize = 2 * n - 4 - colSize * 2;
```

За да отпечатаме на конзолата **покрива**, ще използваме метода **repeat(n)**, която съединява даден стринг **n** на брой пъти.

```
let towerTop = "/" + "^".repeat(colSize) + "\\";
console.log(towerTop + "_" .repeat(midSize) + towerTop);
```



\\ е специален символ в езика **JavaScript** и използвайки само него в метода **console.log(...)**, 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**”).

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 (let row = 0; row < n-3; row++) {
    console.log("|" + " ".repeat(2 * n - 2) + "|");
}
```

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 декларирани от нас променливи **colSize** и **midSize**, защото от примерите виждаме, че са равни на броя **\_** в покрива.

```
console.log("|" +
" ".repeat(colSize+1) +
"_".repeat(midSize) +
" ".repeat(colSize+1) +
 "|");
```

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 `/`.

```
let towerBottom = "\\\" + "_" .repeat(colSize) + "/";
console.log(towerBottom + " ".repeat(midSize) + towerBottom);
```

## Тестване в Judge системата

Тествайте решението си: <https://judge.softuni.bg/Contests/Practice/Index/936#0>.

## Задача: пеперуда

Да се напише програма, която приема **цяло число n** и чертае **пеперуда** с ширина  $2 * n - 1$  колони и височина  $2 * (n - 2) + 1$  реда като в примерите по-долу. **Лявата** и **дясната** ѝ част са широки  $n - 1$ .

### Примерен вход и изход

Вход	Изход	Вход	Изход	Вход	Изход
3	*\ /* @ */ \*	5	***\ /*** ---\ /--- ***\ /*** @ ***/ \*** ---/ \--- ***/ \***	7	*****\ /***** ----\ /---- *****\ /***** ----\ /---- *****\ /***** @ *****/ \***** ----/ \---- *****/ \***** ----/ \---- *****/ \*****

### Входни данни

Входът се състои от един елемент (аргумент) - **цяло число n** [3 ... 1000].

### Изходни данни

Да се отпечатат на конзолата  $2 * (n - 2) + 1$  текстови реда, изобразяващи пеперудата, точно както в примерите.

### Насоки и подсказки

Аналогично на предходната задача виждаме от условието, че **входните данни** ще се състоят само от едно **цяло число** в интервала [3 ... 1000]. Създаваме функция, която приема като аргумент **масив от един елемент**. Тъй като той е от тип **текст**

(**String**), а ние трябва да работим с числа, използваме конструктора **Number()** като функция, с която да го конвертираме:

```
function butterfly([arg1]){
    let n = Number(arg1);
```

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 горното крило на пеперудата е с големина **n - 2**.

```
let halfRowSize =
```

За да нарисуваме горното крило правим цикъл, който да се повтаря **halfRowSize** пъти:

```
for (let i = 1; i <= halfRowSize; i++) {  
}
```

От примерите можем също така да забележим, че на **четен** ред имаме начало **\***, среда **\ /** и край **\***, а на **нечетен** - начало **-**, среда **\ /** и край **-**. Следователно при всяка итерация на цикъла трябва да направим **if-else** проверка дали редът, който печатаме, е четен или нечетен. От примерите, дадени в условието, виждаме, че броят на звездичките и тиретата на всеки ред също е равен на **n - 2**, т. е. за тяхното отпечатване отново можем да използваме променливата **halfRowSize**.

```
for (let i = 0; i < halfRowSize; i++) {
    if(i % 2 == 1){
        console.log("-".repeat(halfRowSize) +
                    "\\" +  
                    " " +  
                    "/" +  
                    "-".repeat(halfRowSize));
    } else {  
        console.log("-".repeat(halfRowSize) +
                    "-" +  
                    " " +  
                    "-" +  
                    "-".repeat(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**). От примерите виждаме, че броят на празните места е **n-1**.

```
console.log(" ".repeat(n - 1) + "@" + " ".repeat(n - 1));
```

Остава да отпечатаме на конзолата и **долното крило**, което е **аналогично на горното крило**: единствено трябва да разменим местата на наклонените черти.

## Тестване в Judge системата

Тествайте решението си: <https://judge.softuni.bg/Contests/Practice/Index/936#1>.

### Задача: знак "Стоп"

Да се напише програма, която приема **цяло число n** и чертае **предупредителен знак STOP** с размери като в примерите по-долу.

#### Входни данни

Входът е състоящ се от един елемент (аргумент) - **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

Както при предните задачи, създаваме функция, която приема масив от един елемент и с **Number()** го преобразуваме от тип текст (**String**) към число:

```
function stopSign([arg1]){
    let n = Number(arg1);
```

Можем да **разделим** фигурата на 3 части - горна, средна и долната. **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**.

```
let dots = .. *
```

Трябва да създадем и втора **променлива**, в която ще пазим **стойността** на **средата** на **началния ред** с големина **`2 * n + 1`**.

```
let underscores = 2 * n + 1;
```

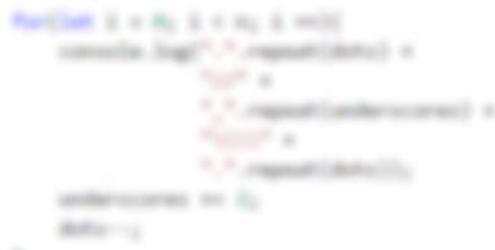
След като вече сме декларирали и инициализирали двете променливи, можем да отпечатаме на конзолата началния ред.

```
console.log(".".repeat(dots) +
    "_".repeat(underscores) +
    ".".repeat(dots));
```

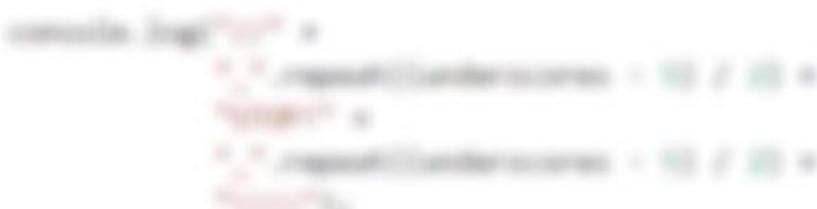
За да начертаем редовете, в които знака се "разширява", трябва да създадем **цикъл**, който да се завърти **`n`** брой пъти. Структурата на един ред се състои от начало `.`, `//` + среда `_` + `\\"` и край `..`. За да можем да използваме отново създадените **променливи**, трябва да намалим **`dots`** с 1 и **`underscores`** с 2, защото ние вече сме **отпечатали** първия ред, а точките и долните черти в горната част от фигурата на всеки ред **намаляват**.

```
underscores -= 2;
dots--;
```

На всяка следваща итерация **началото** и **крайт** намаляват с 1, а **средата** се увеличава с 2.



Средната част от фигурата има начало `// + _`, среда **STOP!** и край `_ + \\  
\\`. Броят на долните черти `_` е **(underscores - 5) / 2**.



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 можем да кажем, че броят на **точките в долната част от фигурата** е равен на `i`.

За да работи нашата програма правилно, трябва на всяка итерация от **цикъла** да **намаляваме** броя на `_` с 2.

```
for(let i = 0; i < n; i++){
    console.log(".".repeat(i) +
        "\\\\" +
        "_" .repeat(underscores) +
        "//" +
        ".".repeat(i));
    underscores -= 2;
}
```

## Тестване в Judge системата

Тествайте решението си: <https://judge.softuni.bg/Contests/Practice/Index/936#2>.

## Задача: стрелка

Да се напише програма, която приема **цяло нечетно число n** и чертае **вертикална стрелка** с размери като в примерите по-долу.

### Входни данни

Входът се състои от **цяло нечетно число n** (аргумент) в интервала [3 ... 79].

### Изходни данни

Да се отпечата на конзолата вертикална стрелка, при която "#" (диез) очертава стрелката, а "." - останалото.

## Примерен вход и изход

Вход	Изход	Вход	Изход
3	.###. .#.#. . ##.## .#.#. . ..#..	5	..#####.. ..#....#.. ..#....#.. ..#....#.. ####...### .#.....#. . ..#....#.. ....#.###. ....#....

## Насоки и подсказки

Както при предните задачи, създаваме функция, която приема масив от един елемент и с **Number()** го преобразуваме от текст към число:

```
function drawArrow([arg1]){
    let n = Number(arg1);
```

Можем да разделим фигурата на **3 части** - горна, средна и долна. **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**:

```
let 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:

```
let innerDots = n - 2;
```

От примерите можем да видим структурата на началния ред. Трябва да използваме декларирани и инициализирани от нас **променливи outerDots** и **n**, за да отпечатаме **началния ред**:

```
console.log(".".repeat(outerDots) +
    "#".repeat(n) +
    ".".repeat(outerDots));
```

За да нарисуваме на конзолата **тялото на стрелката**, трябва да създадем **цикъл**, който да се повтори **n - 2** пъти:

```

for (let i = 0; i < n - 2; i++){
    console.log(".".repeat(outerDots) +
        "#" +
        ".".repeat(innerDots) +
        "#" +
        ".".repeat(outerDots));
}

```

Средата на фигурата е съставена от начало **#**, среда **.** и край **#**. Броят на **#** е равен на **outerDots + 1**:

```

console.log("#".repeat(outerDots + 1) +
    ".".repeat(innerDots) +
    "#".repeat(outerDots + 1));

```

За да начертаем **долната част на стрелката**, трябва да зададем нови стойности на двете **променливи outerDots** и **innerDots**:

```

outerDots = 1;
innerDots = 2 * n - 5;

```

При всяка итерация **outerDots** се увеличава с 1, а **innerDots** намалява с 2. Забелязваме, че тъй като на предпоследния ред стойността на **innerDots** ще е 1 при последвала итерация на цикъла тя ще стане **отрицателно число**. Ако използваме **метода str.repeat(count)** с отрицателно число, програмата ни **ще даде грешка**. Един вариант да избегнем това е да отпечатаме последния ред на фигурата отделно.

Височината на **долната част на стрелката** е **n - 1**, следователно **цикълът**, който ще отпечатва всички редове без последния, трябва да се завърти **n - 2** пъти:

```

for (let i = 0; i < n - 2; i++){
    console.log(".".repeat(outerDots) +
        "#" +
        ".".repeat(innerDots) +
        "#" +
        ".".repeat(outerDots));

    outerDots++;
    innerDots -= 2;
}

```

**Последният ред** от нашата фигура е съставен от начало **.**, среда **#** и край **..**. Броят на **.** е равен на **outerDots**:

```
console.log(".".repeat(outerDots) +
    "#" +
    ".".repeat(outerDots));
```

## Тестване в Judge системата

Тествайте решението си: <https://judge.softuni.bg/Contests/Practice/Index/936#3>.

## Задача: брадва

Да се напише програма, която приема **цяло число n** и чертае брадва с размери, показани по-долу. Ширината на брадвата е  $5 * n$  колони.

### Входни данни

Входът се състои от един елемент (аргумент) - **цяло число n** в интервала [2..42].

### Изходни данни

Да се отпечата на конзолата **брадва**, точно както е в примерите.

### Примерен вход и изход

Вход	Изход	Вход	Изход
2	<pre>-----**----- -----*-*- *****-*_- -----***-</pre>	5	<pre>-----**----- -----*-*- -----*-*- -----*-*- -----*-*- -----*-*- *****-*_- *****-*_- -----*-*- -----*****-</pre>

## Насоки и подсказки

За решението на задачата е нужно първо да изчислим големината на **ти retata отляво**, **средните тирета**, **ти retata отдясно** и **цялата дължина на фигурата**.

```
let width = 5 * n;
let leftDashes = 3 * n;
let middleDashes = 0;
let 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** на брой пъти. При всяка итерация от цикъла **средните тирета** се увеличават с 1, а **тиретата отляво** се намаляват с 1.

```
for (let i = 0; i < n; i++){
    console.log("-".repeat(leftDashes) +
        "*" + "-".repeat(middleDashes) +
        "*" + "-".repeat(rightDashes));

}
```

Сега следва да нарисуваме **дръжката на брадвата**. За да можем да използваме отново създадените **променливи** при чертането на дръжката на брадвата, трябва да намалим **средните тирета** с 1, а **тези отляво и отляво** да увеличим с 1.

```
middleDashes--;
rightDashes++;
leftDashes++;
```

**Дръжката на брадвата** можем да нарисуваме, като завъртим цикъл, който се повтаря

**n / 2** пъти. Можем да отделим тази стойност в отделна **променлива**, като внимаваме, че ако като входни данни имаме **нечетно число**, при деление на 2 резултатът ще е **реално число** с цяла и дробна част. Тъй като в този случай ни трябва **само цялата част** (от условието на задачата виждаме, че при вход **5** височината на дръжката на брадвата е **2**), можем да използваме метода **Math.trunc()**, с който да запазим само нейната стойност в новата ни променлива.

От примерите можем да разберем, каква е структурата на дръжката:

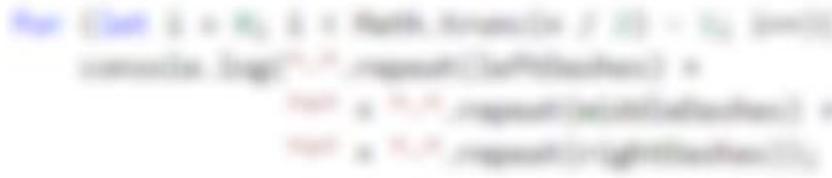
```
axeHeight = Math.trunc(n / 2);

for (let i = 0; i < axeHeight; i++) {
    console.log("*".repeat(leftDashes) +
        "-".repeat(middleDashes) + "*" +
        "-".repeat(rightDashes));
}
```

**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 **axeHeight - 1** пъти. На всяка итерация тиретата отляво и тиретата отдясно намаляват с 1, а средните тирета се увеличават с 2.

```
leftDashes --;
```



```
middleDashes += 2;  
leftDashes--;  
rightDashes--;  
}
```

За последния ред от фигурата, можем отново да използваме трите, вече декларириани и инициализирани променливи **leftDashes**, **middleDashes**, **rightDashes**.

```
console.log("-".repeat(leftDashes) +  
          "*" + "*".repeat(middleDashes) +  
          "*" + "-".repeat(rightDashes));
```

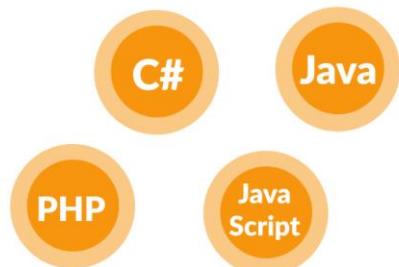
## Тестване в Judge системата

Тествайте решението си: <https://judge.softuni.bg/Contests/Practice/Index/936#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](http://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.

## Видео

Гледайте видео-урок по тази глава тук: <https://youtu.be/fDVHaakZODU>.

## 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 обърнем **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.

Най-често 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**. При стъпка 3, цикълът би изглеждал по следния начин:

```
let n = parseInt(arg1);
for (let i = 1; i < n; i += 3) {
    console.log(i);
}
```

Задаване  
на стъпка

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):

- Създаваме функция, която ще приема числото  $n$ .
- Изпълняваме **for цикъл** от 1 до  $n$  с размер на стъпката 3.
- В **тялото на цикъла** отпечатваме стойността на текущата стъпка.

```
// чрез i+=3 повишаваме стойността на i с размера на стъпката
function loopByStep3([arg1]) {
    let n = parseInt(arg1);
    for (let i = 1; i <= n; i+=3) {
        console.log(i);
    }
}
```

## Тестване в Judge системата

Тествайте решението си: <https://judge.softuni.bg/Contests/Practice/Index/937#0>.

## Пример: числата от N до 1 в обратен ред

Да се напише програма, която отпечатва числата от  $n$  до 1 в обратен ред (стъпка -1). Например, ако  $n = 100$ , то резултатът ще е: 100, 99, 98, ..., 3, 2, 1.

Можем да решим задачата по следния начин:

- Създаваме функция, която ще приема числото  $n$ .
- В него изпълняваме **for цикъл**, като присвояваме **let i = n**.
- Обръщаме условието на цикъла: **i >= 1**.
- Дефинираме размера на стъпката: **-1**.
- В **тялото на цикъла** отпечатваме стойността на текущата стъпка.

```
// Обърнато условие: i >= 1
// Намаляваща стъпка: i -= 1
function numbersNto1([arg1]) {
    let n = parseInt(arg1);
    for (let i = n; i >= 1; i-=1) {
        console.log(i);
    }
}
```

## Тестване в Judge системата

Тествайте решението си: <https://judge.softuni.bg/Contests/Practice/Index/937#1>.

## Пример: числата от 1 до $2^n$ с for цикъл

В следващия пример ще разгледаме как се ползва обичайната стъпка с размер 1.

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.

```
function powersOfTwo([arg1]) {
    let n = parseInt(arg1);
    let num = 1;

    for (let i = 0; i <= n; i++) {
        console.log(num);
        num = num * 2;
    }
}
```

Тестване в Judge системата

Тествайте решението си: <https://judge.softuni.bg/Contests/Practice/Index/937#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 как можем да решим задачата:

- Създаваме функция, която ще приема числото **n**.
- Декларираме 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).

```
function evenPowersOfTwo([arg1]) {
    let n = parseInt(arg1);
    let num = 1;

    for (let i = 0; i <= n; i+=2) {
        console.log(num);
        num = num * 2 * 2;
    }
}
```

Тестване в Judge системата

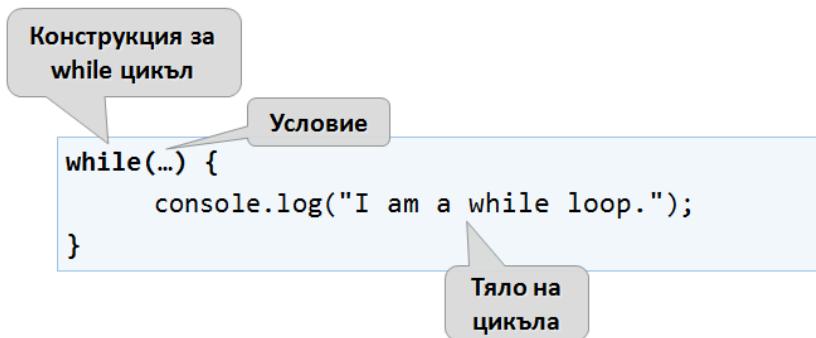
Тествайте решението си: <https://judge.softuni.bg/Contests/Practice/Index/937#3>.

## 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, но имат по-опростен синтаксис.

### Какво представлява while цикълът?

В програмирането **while** цикълът се използва, когато искаме да повторяме извършването на определена логика, докато е в сила дадено условие. Под "условие", разбираем всеки израз, който връща **true** или **false**. Когато условието стане грешно, **while** цикълът прекъсва изпълнението си и програмата продължава с изпълняването на кода 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 функция, която ще приема числото **n**.
- Декларираме 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 променливата и я увеличаваме, използвайки формулата от условието на задачата.

Ето и примерна реализация на описаната идея:

```
function sequence([arg1]) {
    let n = parseInt(arg1);
    let num = 1;

    while (num <= n) {
        console.log(num);
        num = 2 * num + 1;
    }
}
```

### Тестване в Judge системата

Тествайте решението си: <https://judge.softuni.bg/Contests/Practice/Index/937#4>.

### Пример: число в диапазона [1 ... 100]

Да се въведе цяло число в диапазона [1 ... 100]. Ако то 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:

- Декларираме променлива **i**, на която присвояваме начална стойност **0**. Чрез нея ще запазваме позицията на всяко число, което е подадено на функцията.
- Декларираме променлива **num**, на която присвояваме целочислената стойност на първия аргумент, подаден на функцията.
- За условие на цикъла слагаме израз, който е **true**, ако числото **не е** в диапазона, посочен в условието.
- В **тялото на цикъла**: увеличаваме **i**, за да може, при следващото завъртане на цикъла, да вземем следващото число, което е подадено на функцията. Отпечатваме съобщение със съдържание "**Invalid number!**" на конзолата, след което присвояваме нова стойност за **num** (следващия аргумент, подаден на функцията).
- След като вече сме валидирали числото, извън тялото на цикъла отпечатваме стойността му.

Ето и примерна реализация на алгоритъма чрез **while** цикъл:

```
function numberInRange(args) {
    let i = 0;
    let num = parseInt(args[i]);
```

```

while (num < 1 || num > 100) {
    i++;
    console.log("Invalid number!");
    num = parseInt(args[i]);
}

console.log(`The number is: ${num}`);
}

```

## Тестване в Judge системата

Тествайте решението си: <https://judge.softuni.bg/Contests/Practice/Index/937#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 се дели **едновременно** и на **a**, и на **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;

```

## Пример: най-голям общ делител (НОД)

Да се подадат **цели** числа **a** и **b** и да се намери **НОД(a, b)**.

Ще решим задачата чрез **алгоритъма на Евклид**:

- Декларираме променливи **a** и **b**, на които присвояваме **целочислени** стойности, подадени на функцията.
- За условие на цикъла слагаме израза **b ≠ 0**.
- В **тялото на цикъла** следваме указанията от псевдо кода:
  - Декларираме временна променлива, на която присвояваме **текущата** стойност на **b**.
  - Присвояваме нова стойност на **b** – остатъкът 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.

```
function greatestCommonDivisor([arg1, arg2]) {
  let a = parseInt(arg1);
  let b = parseInt(arg2);

  while (b !== 0) {
    let oldB = b;
    b = a % b;
    a = oldB;
  }

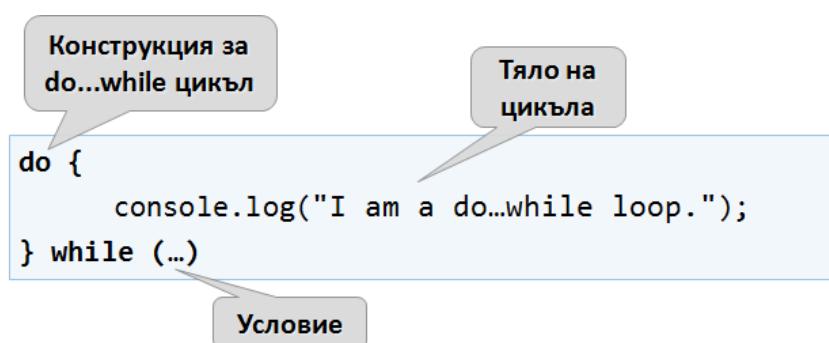
  console.log("GCD = " + a);
}
```

### Тестване в Judge системата

Тествайте решението си: <https://judge.softuni.bg/Contests/Practice/Index/937#6>.

## Do-while цикъл

Следващият цикъл, с който ще се запознаем, е **do-while**, в превод - **прави-докато**. 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, чиито 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:

- Декларираме променливата **n**, на която присвояваме целочислена стойност подадена на функцията.
- Създаваме още една променлива - **fact**, чиято начална стойност е 1. Няя 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.

```

function factorial([arg1]) {
  let n = parseInt(arg1);
  let fact = 1;

  do {
    fact = fact * n;
    n--;
  } while (n > 1);

  console.log(fact);
}

```

## Тестване в Judge системата

Тествайте решението си: <https://judge.softuni.bg/Contests/Practice/Index/937#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:

- Декларираме 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.

```
function sumDigits([arg1]) {
  let n = parseInt(arg1);
  let sum = 0;

  do {
    sum = sum + (n % 10);
    n = Math.floor(n / 10);
  } while (n > 0);

  console.log("Sum of digits: " + sum);
}
```



- **n % 10**: връща последната цифра на числото **n**.
- **Math.floor(n / 10)**: изтрива последната цифра на **n**.

## Тестване в Judge системата

Тествайте решението си: <https://judge.softuni.bg/Contests/Practice/Index/937#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.log("Infinite loop");
}
```

А така изглежда **безкраен for** цикъл:

```
for (;;) {
    console.log("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 изисква 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 едно 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:

- $10 = 2 * 5$
- $42 = 2 \cdot 3 \cdot 7$
- $143 = 13 * 11$

**Алгоритъм за проверка** дали дадено цяло число е **просто**: проверяваме дали  $n > 1$  и дали  $n$  се дели на  $2, 3, \dots, n-1$  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  $\sqrt{n}$ . Помислете 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 алгоритъма за проверка за просто число, разписан постъпково:

- Декларираме променливата `n`, на която присвояваме цялото число, което е подадено на функцията.
- Създаваме булева променлива `prime` с начална стойност `true`. Приемаме, че едно число е просто до доказване на противното.
- Създаваме `for` цикъл, чиято 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 `prime` на `false` и излизаме принудително от цикъла чрез оператор `break`.
- В зависимост от стойността на `prime` отпечатваме дали числото е просто (`true`) или съответно (`false`).

Ето и примерна имплементация на описания алгоритъм:

```
function isPrime([arg1]) {
  let n = parseInt(arg1);
  let prime = true;

  for (let i = 2; i <= Math.sqrt(n); i++) {
```

```

if (n % i === 0) {
    prime = false;
    break;
}
}

if (prime && n > 2) {
    console.log("Prime");
} else {
    console.log("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/937#9>.

## Пример: оператор `break` в безкраен цикъл

Да се напише програма, която проверява дали едно число `n` е четно, 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:

- Декларираме променлива `i`, на която присвояваме начална стойност 0. Чрез нея ще запазваме позицията на всяко число, което е подадено на функцията.
- Декларираме променлива `num`, на която присвояваме начална стойност 0.
- Създаваме безкраен `while` цикъл, като за условие ще зададем `true`.
- В **тялото на цикъла**:
  - Вземаме целочислена стойност, която е подадена на функцията и я присвояваме на `num`.
  - Ако **числото е четно**, излизаме от цикъла чрез `break`.
  - В **противен случай** извеждаме съобщение, което гласи, че **числото не е четно**. Увеличаваме `i`, за да може, при следващото завъртане на цикъла, да вземем следващото число, което е подадено на функцията. Итерациите продължават, докато не се въведе четно число.
- Отпечатваме четното число на екрана.

Ето и примерна имплементация на идеята:

```

function enterEvenNumber(args) {
    let i = 0;
    let num = 0;

    while (true) {
        num = parseInt(args[i]);

        if (num % 2 == 0) {
            break; // even number -> exit from the loop
        }

        console.log("The number is not even.");
        i++;
    }

    console.log(`Even number entered: ${num}`);
}

```

## Тестване в Judge системата

Тествайте решението си: <https://judge.softuni.bg/Contests/Practice/Index/937#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:

```

1 3
1 2

```

Ето едно **грешно решение**, което изглежда правилно на пръв поглед:

```

function combinations() {
    for (let i = 1; i <= 3; i++) {
        for (let j = 3; j >= 1; j--) {

```

```

    if (i + j === 2){
        break;
    }
    console.log(i + " " + j);
}
}
}

```

Ако оставим програмата ни по този начин, резултатът ни ще е следният:

```

1 3
1 2
2 3
2 2
2 1
3 3
3 2
3 1

```

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 продължава въртенето на цикъла. При нужда от изход (излизане от всички вложени цикли), променливата сменя стойността си на **true** и се излиза от вътрешния цикъл с **break**, а при последваща проверка се напуска и външния цикъл. Ето и примерна имплементация на тази идея:

```

function combinations() {
    let hasToEnd = false;

    for (let i = 1; i <= 3; i++) {
        if (hasToEnd === false) {
            for (let j = 3; j >= 1; j--) {
                if (i + j === 2) {

```

```

    hasToEnd = true;
    break;
}

console.log(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 цикъл и ще прекъсне изпълнението си.

## Тестване в Judge системата

Тествайте решението си: <https://judge.softuni.bg/Contests/Practice/Index/937#11>.

## 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 Фиbonачи

Числата на Фибоначи 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 Фибоначи.

## Насоки и подсказки

Идея за решаване на задачата:

- Декларираме променлива **n**, на която присвояваме целочислена стойност, подадена на функцията.
- 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 Фиbonачи.
  - На **f0** присвояваме текущата стойност на **f1**.
  - На **f1** присвояваме стойността на временната променлива **fNext**.
- Извън цикъла отпечатваме числото **n**-тото число на Фиbonачи.

Примерна имплементация:

```
function fibonacci([arg1]) {
  let n = Number(arg1);
  let f0 = 1;
  let f1 = 1;

  for (let i = 0; i < n - 1; i++) {
    let fNext = f0 + f1;
    f0 = f1;
    f1 = fNext;
  }

  console.log(f1);
}
```

## Тестване в Judge системата

Тествайте решението си: <https://judge.softuni.bg/Contests/Practice/Index/937#12>.

## 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	Вход	Изход	Вход	Изход
7	1 2 3 4 5 6 7	5	1 2 3 4 5	10	1 2 3 4 5 6 7 8 9 10

## Насоки и подсказки

Можем да решим задачата с **два вложени цикъла** (по редове и колони) с печатане в тях и излизане при достигане на последното число. Ето идеята, разписана по-подробно:

- Декларираме променлива **n**, на която присвояваме целочислена стойност, подадена на функцията.
- Декларираме променлива **num** с начална стойност 1. Тя ще пази броя на отпечатаните числа. При всяка итерация ще я **увеличаваме** с **1** и ще я добавяме към текущия ред.
- Декларираме променлива **result**, която ще е текущият ред и към която ще добавяме стойността на текущата клетка.
- Създаваме **външен for** цикъл, който ще отговаря за **редовете** в таблицата. Наименуваме променливата на цикъла **row** и ѝ задаваме начална стойност 1. 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 1. За условие слагаме **col < row** (**row** = брой цифри на ред). Размерът на стъпката е 1.
- В тялото на вложениния цикъл:
  - Проверяваме дали **col > 1**, ако да – добавяме разстояние към променливата **result**. Ако не направим тази проверка, а директно добавяме разстоянието, ще имаме ненужно такова в началото на всеки ред.
  - **Запазваме** 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 стойността на променливата **result**, след което ѝ задаваме нова празна стойност. По този начин ще преминем на следващия ред.
- Отново проверяваме дали **num > n**. Ако е по-голямо, прекъсваме изпълнението на **програмата ни** чрез **break**.

Ето и примерна имплементация:

```

function numberPyramid([arg1]) {
    let n = parseInt(arg1);
    let num = 1;
    let result = "";

    for (let row = 1; row <= n; row++) {
        for (let col = 1; col <= row; col++) {
            if (col > 1) {
                result += " ";
            }

            result += num;
            num++;
        }

        if (num > n) {
            break;
        }
    }

    console.log(result);
    result = "";
}

if (num > n) {
    break;
}
}
}
}

```

## Тестване в Judge системата

Тествайте решението си: <https://judge.softuni.bg/Contests/Practice/Index/937#13>.

## 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:

- Взимаме размера на таблицата от целочислената променлива **n**, която е подадена на функцията.
- Декларираме променлива **result**, която ще е текущият ред и към която ще добавяме стойността на текущата клетка.
- 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 вложениния 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 с цел да не превишаваме **n** в никоя от клетките на таблицата.
    - Добавяме числото от текущата клетка в променливата **result**.
- Отпечатваме стойността на **result**, след което ѝ задаваме нова празна стойност. По този начин ще преминем на следващия ред.

```
function numberTable([arg1]) {
  let n = parseInt(arg1);
  let result = "";

  for (let row = 0; row < n; row++) {
    for (let col = 0; col < n; col++) {
      let num = row + col + 1;

      if (num > n) {
        num = 2 * n - num;
      }

      result = result + num + " ";
    }
  }
}
```

```

        console.log(result);
        result = "";
    }
}

```

## Тестване в Judge системата

Тествайте решението си: <https://judge.softuni.bg/Contests/Practice/Index/937#14>.

## Какво научихме от тази глава?

Можем да използваме **for** цикли със стъпка:

```

for (let i = 1; i <= n; i+=3) {
    console.log(i);
}

```

Циклите **while** / **do-while** се повтарят докато е в сила дадено **условие**:

```

let num = 1;
while (num <= n) {
    console.log(num++);
}

```

Ако се наложи да прекъснем изпълнението на цикъл, го правим с оператора **break**:

```

let n = 0;
while (true) {
    n = parseInt(arg1);
    if (n % 2 === 0) {
        break; // even number -> exit from the loop
    }
    console.log("The number is not even.");
}
console.log(`Even number entered: ${n}`);

```

# 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.

### Задача: генератор за тъпи пароли

Да се напише програма, която въвежда две цели числа **n** и **k** и генерира по азбучен ред всички възможни "тъпи" пароли<sup>1</sup>, 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 **k** букви на латинската азбука.
- Символ 4: малка буква измежду първите **k** букви на латинската азбука.
- Символ 5: цифра от 1 до **n**, по-голяма от първите 2 цифри.

### Входни данни

Входът е масив от **две цели числа** (аргумента): **n** и **k** в интервала [1 ... 9]

### Изходни данни

На конзолата трябва да се отпечатат всички "тъпи" пароли по азбучен ред, разделени с **интервал**.

### Примерен вход и изход

Вход	Изход	Вход	Изход
2	11aa2 11ab2 11ac2 11ad2 11ba2 11bb2 11bc2 11bd2 11ca2 11cb2 11cc2 11cd2	3	11aa2 11aa3 12aa3 21aa3 22aa3
4	11da2 11db2 11dc2 11dd2	1	
Вход	Изход	Вход	Изход
4	11aa2 11aa3 11aa4 11ab2 11ab3 11ab4 11ba2 11ba3 11ba4 11bb2 11bb3 11bb4 12aa3 12aa4 12ab3 12ab4 12ba3 12ba4 12bb3 12bb4 13aa4 13ab4 13ba4 13bb4 21aa3 21aa4 21ab3 21ab4	3	11aa2 11aa3 11ab2 11ab3 11ba2 11ba3 11bb2 11bb3
2	21ba3 21ba4 21bb3 21bb4 22aa3 22aa4 22ab3 22ab4 22ba3 22ba4 22bb3 22bb4 23aa4 23ab4 23ba4 23bb4 31aa4 31ab4 31ba4 31bb4 32aa4 32ab4 32ba4 32bb4 33aa4 33ab4 33ba4 33bb4	2	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 и конвертиране на входните данни** – в настоящата задача това включва вземането на два елемента от подадения масив **n** и **k** и преобразуването им в числа.
- **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 константи **const: n** и **k**, освен това ще създадем променлива **solution**, в която ще пазим **string** с отговора.

```
function passwordGenerator(input) {
    const n = Number(input[0]);
    const k = Number(input[1]);
    let solution = '';
    // TODO: password generator logic

    return solution;
}
```

## Извеждане на резултат

Един от начините да намерим решението на тази задача е да създадем пет **for** цикъла, вложени един в друг, по един за всеки символ. За да гарантираме условието последния символ, който по условие е число, да бъде **по-голям** от първите два, които също са числа, ще използваме вградения метод **Math.max(...)**:

```
function passwordGenerator(input) {
    const n = Number(input[0]);
    const l = Number(input[1]);
    let solution = '';

    for (var s1 = 1; s1 <= n; s1++) {
        for (var s2 = 1; s2 <= n; s2++) {
            for (var s3ascii = 97; s3ascii < 97 + 1; s3ascii++) {
                let s3 = String.fromCharCode(s3ascii);

                for (var s4ascii = 97; s4ascii < 97 + 1; s4ascii++) {
                    let s4 = String.fromCharCode(s4ascii);

                    for (var s5 = Math.max(s1, s2) + 1; s5 <= n; s5++) {
                        solution = '${s1}${s2}${s3}${s4}${s5}';
                    }
                }
            }
        }
    }

    return solution;
}
```

## Тестване в Judge системата

Тествайте решението си: <https://judge.softuni.bg/Contests/Practice/Index/938#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 →  $1 * 1 * 1 * 1 * 1 * 2 = 2$

- $111121 \rightarrow 1 * 1 * 1 * 1 * 2 * 1 = 2$
- $111211 \rightarrow 1 * 1 * 1 * 2 * 1 * 1 = 2$
- $112111 \rightarrow 1 * 1 * 2 * 1 * 1 * 1 = 2$
- $121111 \rightarrow 1 * 2 * 1 * 1 * 1 * 1 = 2$
- $211111 \rightarrow 2 * 1 * 1 * 1 * 1 * 1 = 2$

## Входни данни

Програмата прочита **едно цяло число** (аргумент) в интервала [1 ... 600 000].

## Изходни данни

На конзолата трябва да се отпечатат **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	53144 1	999999

## Насоки и подсказки

Решението на задачата за магическите числа следва **същата** концепция (отново трябва да генерираме всички комбинации за n елемента). Следвайки следните стъпки, опитайте да решите задачата сами:

- Инициализирайте **променлива** в която ще запазите стойността на магическото число.
- 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 произведението на шестте цифри е **равно** на **магическото** число.

```
function magincNumber(input) {
  const magicNum = Number(input);
  let solution = '';

  for (var d1 = 1; d1 < 10; d1++) {
    for (var d2 = 1; d2 < 10; d2++) {
      for (var d3 = 1; d3 < 10; d3++) {
        for (var d4 = 1; d4 < 10; d4++) {
          for (var d5 = 1; d5 < 10; d5++) {
            for (var d6 = 1; d6 < 10; d6++) {
              if (d1 * d2 * d3 * d4 * d5 * d6 === magicNum) {
                solution =
                  solution.concat(d1, d2, d3, d4, d5, d6, ' ');
              }
            }
          }
        }
      }
    }
  }

  return solution;
}
```

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.

```
function magincNumber(input) {
  const magicNum = Number(input);
  let solution = '';

  // TODO: magicNumber generator logic

  return solution;
}
```

След това ще започнем да разписваме **while** циклите.

- Ще инициализираме **първата цифра**: **d1 = 1**.
- Ще зададем **условие за всеки цикъл**: цифрата да бъде по-малка от 10.
- В **началото** на всеки цикъл задаваме стойност на **следващата цифра**, в случая: **d2 = 1**. 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 добавим резултата към променливата съхраняваща решението.

```
function magicNumber (input) {
    const magicNum = Number(input)
    let d1, d2, d3, d4, d5, d6;

    let solution = '';
    d1 = 1;
    while (d1 < 10) {
        d2 = 1;
        while (d2 < 10) {
            d3 = 1;
            while (d3 < 10) {
                d4 = 1;
                while (d4 < 10) {
                    d5 = 1;
                    while (d5 < 10) {
                        d6 = 1;
                        while (d6 < 10) {
                            if (d1 * d2 * d3 * d4 * d5 * d6 === magicNum) {
                                solution =
                                    solution.concat(d1, d2, d3, d4, d5, d6, ' ');
                            }
                            d6++;
                        }
                        d5++;
                    }
                    d4++;
                }
                d3++;
            }
            d2++;
        }
        d1++;
    }
}
```

```

    d2++;
}
d1++;
}
return solution;
}

```

Както виждаме, един проблем можем да решим с различни видове цикли. Разбира се, за всяка задача има най-подходящ избор. С цел да упражните всеки от циклите, 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/938#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.

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 (let 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/938#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, който програмата чете, се състои от **едно цяло число** (аргумент) в интервала [1 ... 600 000].

### 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

Вход	Изход	Вход	Изход
11	1111	16	1111 1112 1114 1118 1121 1122 1124 1128 1141 1142 1144 1148 1181 1182 1184 1188 1211 1212 1214 1218 1221 1222 1224 1228 1241 1242 1244 1248 1281 1282 1284 1288 1411 1412 1414 1418 1421 1422 1424 1428 1441 1442 1444 1448 1481 1482 1484 1488 1811 1812 1814 1818 1821 1822 1824 1828 1841 1842 1844 1848 1881 1882 1884 1888 2111 2112 2114 2118 2121 2122 2124 2128 2141 2142 2144 2148 2181 2182 2184 2188 2211 2212 2214 2218 2221 2222 2224 2228 2241 2242

Вход	Изход	Вход	Изход
			2244 2248 2281 2282 2284 2288 2411 2412 2414 2418 2421 2422 2424 2428 2441 2442 2444 2448 2481 2482 2484 2488 2811 2812 2814 2818 2821 2822 2824 2828 2841 2842 2844 2848 2881 2882 2884 2888 4111 4112 4114 4118 4121 4122 4124 4128 4141 4142 4144 4148 4181 4182 4184 4188 4211 4212 4214 4218 4221 4222 4224 4228 4241 4242 4244 4248 4281 4282 4284 4288 4411 4412 4414 4418 4421 4422 4424 4428 4441 4442 4444 4448 4481 4482 4484 4488 4811 4812 4814 4818 4821 4822 4824 4828 4841 4842 4844 4848 4881 4882 4884 4888 8111 8112 8114 8118 8121 8122 8124 8128 8141 8142 8144 8148 8181 8182 8184 8188 8211 8212 8214 8218 8221 8222 8224 8228 8241 8242 8244 8248 8281 8282 8284 8288 8411 8412 8414 8418 8421 8422 8424 8428 8441 8442 8444 8448 8481 8482 8484 8488 8811 8812 8814 8818 8821 8822 8824 8828 8841 8842 8844 8848 8881 8882 8884 8888

## Насоки и подсказки

Решете задачата самостоятелно използвайки наученото от предишните две. Спомнете си разликата между операторите за **целочислено деление ( / )** и **деление с остатък (%)** в JavaScript.

## Тестване в Judge системата

Тествайте решението си: <https://judge.softuni.bg/Contests/Practice/Index/938#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, който прочита програмата, е **едно цяло число** (аргумент) в интервала [100 ... 999].

## Изходни данни

На конзолата трябва 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: 376 → нито на 5, нито на 3 → 376 + 6 → = = 382 → нито на 5, нито на 3 → 382 + 6 = = 388 + 6 = 394 + 6 = 400 → 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 Входното число е 132: 132 → деление на 3 → 132 - 3 = = 129 → деление на 3 → 129 - 3 = = 126 → деление на 3 → 126 - 3 = = 123 → деление на 3 → 123 - 3 = = 120 → деление на 5 → 120 - 1 = ..... 121 → нито на 5, нито на 3 → 121 + 2 = 123

## Насоки и подсказки

Решете задачата **самостоятелно**, използвайки наученото от предишните задачи. 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/938#4>.

# 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 упражним** всички знания, които сте придобили от настоящата книга и през курса "Programing Basics".

## Видео

Гледайте видео-урок по тази глава тук: <https://youtu.be/bnxf9oiDduo>.

## 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.

Как работи тестването в **Judge** системата? В **Judge** системата се влиза с вашия softuni.bg акаунт. Качвате сорс кода и от менюто под него избирате да се изпълни с **JavaScript**. Програмата бива **тествана** с поредица от 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  $[-1000 \dots 1000]$ .
- Гарантирано е, че  $y_2 = y_3$ .

### Изходни данни

Да се отпечата на конзолата лицето на триъгълника.

### Примерен вход и изход

Вход	Изход	Чертеж	Обяснения
5 -2 6 1 1 1	7.5	<p>Diagram showing a triangle with vertices <math>(x_1, y_1) = (5, -2)</math>, <math>(x_2, y_2) = (1, -2)</math>, and <math>(x_3, y_3) = (1, 1)</math>. The base <math>a</math> is 5 (from <math>x_1</math> to <math>x_2</math>) and the height <math>h</math> is 3 (from <math>y_1</math> to <math>y_3</math>). The area is <math>S = a * h / 2 = 5 * 3 / 2 = 7.5</math>.</p>	<p>Страната на триъгълника: <math>a = 6 - 1 = 5</math></p> <p>Височината на триъгълника: <math>h = 1 - (-2) = 3</math></p> <p>Лицето на триъгълника: <math>S = a * h / 2 = 5 * 3 / 2 = 7.5</math></p>

Вход	Изход	Чертеж	Обяснения
4 1 -1 -3 3 -3	8	<p>Diagram showing a triangle with vertices <math>(x_1, y_1) = (4, -3)</math>, <math>(x_2, y_2) = (1, -3)</math>, and <math>(x_3, y_3) = (3, -3)</math>. The base <math>a</math> is 2 (from <math>x_2</math> to <math>x_3</math>) and the height <math>h</math> is 4 (from <math>y_1</math> to <math>y_2</math>). The area is <math>S = a * h / 2 = 4 * 4 / 2 = 8</math>.</p>	<p>Страната на триъгълника: <math>a = 3 - (-1) = 4</math></p> <p>Височината на триъгълника: <math>h = 1 - (-3) = 4</math></p> <p>Лицето на триъгълника: <math>S = a * h / 2 = 4 * 4 / 2 = 8</math></p>

## 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 **x1, y1, x2, y2, x3, y3** в този си ред. Ако не спазваме тази последователност, решението става грешно. Първо пишем кода, който чете подадените данни:

```
x1 = Number(x1);
y1 = Number(y1);
x2 = Number(x2);
y2 = Number(y2);
x3 = Number(x3);
y3 = Number(y3);
```

Трябва да пресметнем **страницата** и **височината** на триъгълника. От примерите, както и от условието **y2 = y3** забелязваме, че едната **страница** винаги е успоредна на хоризонталната ос. Това означава, че нейната **дължина** е равна на дълчината на отсечката между нейните координати **x2** и **x3**, 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 **y1** и **y2** (или **y3**, тъй като са равни). Тъй като не знаем дали винаги **x2** ще е по-голям от **x3**, или **y1** 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.

```
let a = Math.abs(x2 - x3);
let h = Math.abs(y2 - y1);
```

По познатата ни от училище формула за намиране на **лице на триъгълник** ще пресметнем лицето.

```
let s = (a * h) / 2;
```

Единственото, което остава, е да отпечатаме лицето на конзолата.

```
console.log(s);
```

## Тестване в Judge системата

Тествайте решението си: <https://judge.softuni.bg/Contests/Practice/Index/939#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.

## Входни данни

Като параметри на функцията подаваме **3 цели числа**:

- **Броят тухли x**
- **Броят работници w**
- **Вместимостта на количката m**

Всички входни числа са цели и в диапазона [1 ... 1000].

## Изходни данни

Да се отпечата на конзолата **минималният брой курсове**, необходими за превозване на тухлите.

### Примерен вход и изход

Вход	Изход	Обяснения
120 2 30	2	Имаме <b>2</b> работника, всеки вози по <b>30</b> тухли на курс. Общо работниците возят по <b>60</b> тухли на курс. За да превозят <b>120</b> тухли, са необходими точно <b>2</b> курса.

Вход	Изход	Обяснения
355 3 10	12	Имаме <b>3</b> работника, всеки вози по <b>10</b> тухли на курс. Общо работниците возят по <b>30</b> тухли на курс. За да превозят <b>355</b> тухли, са необходими точно <b>12</b> курса: <b>11</b> пълни курса превозват <b>330</b> тухли и последният <b>12-ти</b> курс пренася последните <b>25</b> тухли.

Вход	Изход	Обяснения
5 12 30	1	Имаме <b>5</b> работника, всеки вози по <b>30</b> тухли на курс. Общо работниците возят по <b>150</b> тухли на курс. За да превозят <b>5</b> тухли, е достатъчен само <b>1</b> 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.

```
x = Number(x);
w = Number(w);
m = Number(m);
```

Пресмятаме колко **тухли** носят работниците на един курс.

```
let bricksInOneCourse = w * m;
```

Като разделим общия брой на тухлите, пренесени за **1 курс**, ще получим броя **курсове**, необходими за пренасянето им. Ще използваме метода **Math.ceil(...)**, 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.

```
let totalCourses = Math.ceil(x / bricksInOneCourse);
```

Накрая принтираме резултата на конзолата.

```
console.log(totalCourses);
```

## Тестване в Judge системата

Тествайте решението си: <https://judge.softuni.bg/Contests/Practice/Index/939#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 права е разположена **хоризонтална отсечка**, зададена с **x** координатите на двета си края: **first** и **second**. Точка е разположена **върху** същата хоризонтална права 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.

### Входни данни

Като параметри на функцията подаваме 3 цели числа:

- Числото **first** – **единния край** на отсечката.
- Числото **second** – **другия край** на отсечката.
- Числото **point** – **местоположението** на точката.

Всички входни числа са цели и в диапазона [-1000 ... 1000].

## Изходни данни

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.

```
first = Number(first);
second = Number(second);
point = Number(point);
```

Тъй като не знаем коя **точка** е от ляво и коя е от дясно, ще си направим две променливи, които да ни отбелнязват това. Тъй като **лявата 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.

```

let left = Math.min(first, second);
let right = Math.max(first, second);

let distanceLeft = Math.abs(left - point);
let distanceRight = Math.abs(right - point);

```

По-малкото от двете **разстояния** ще намерим ползвайки **Math.min(...)**.

```
let 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 х координатата ѝ се намира **между тях**. В противен случай, точката се намира **извън линията**. След проверката изкарваме едното от двете съобщения, спрямо това коя проверка е удовлетворена.

```

if (point >= left && point <= right) {
    console.log('in');
}
else {
    console.log('out');
}

```

Накрая принтираме **разстоянието**, намерено преди това.

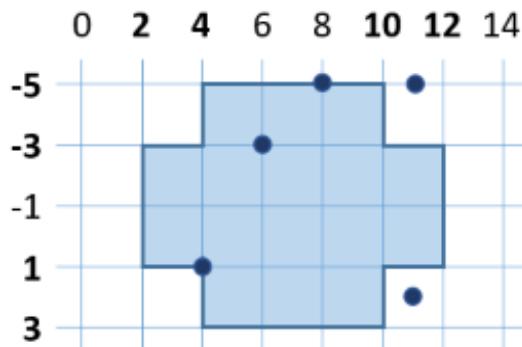
```
console.log(minDistance);
```

### Тестване в Judge системата

Тествайте решението си: <https://judge.softuni.bg/Contests/Practice/Index/939#2>.

### Задача: точка във фигура

Да се напише програма, която проверява дали дадена точка (с координати **x** и **y**) е **вътре** или **извън** следната фигура:



## Входни данни

Като параметри на функцията подаваме **две цели числа: x и y**.

Всички входни числа са цели и в диапазона [-1000 ... 1000].

## Изходни данни

Да се отпечата на конзолата "in" или "out" – дали точката е **вътре** или **извън** фигурата (на контура е вътре).

## Примерен вход и изход

Вход	Изход
8 -5	in

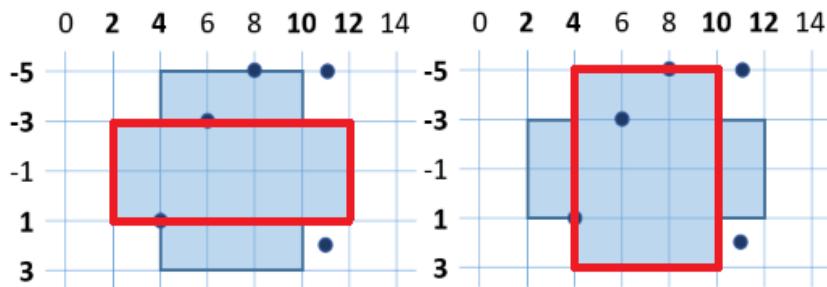
Вход	Изход
6 -3	in

Вход	Изход
11 -5	out

Вход	Изход
11 2	out

## Насоки и подсказки

За да разберем дали **точката** е във **фигурата**, ще разделим **фигурата** на 2 четириъгълника:



Достатъчно условие е **точката** да се намира в един от тях, за да се намира във **фигурата**.

Четем от конзолата входните данни:

```
x = Number(x);
y = Number(y);
```

Ще създадем две променливи, които ще отбелоязват дали **точката** се намира в **някой** от правоъгълниците.

```
let pointInRect1 = x >= 2 && x <= 12 && y >= -3 && y <= 1;
let 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 **фигурата**.

```

if (pointInRect1 || pointInRect2) {
    console.log('in');
}
else {
    console.log('out');
}

```

## Тестване в Judge системата

Тествайте решението си: <https://judge.softuni.bg/Contests/Practice/Index/939#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 отпечатат с **водеща нула**, когато са едноцифриeni (например 01, 05, 08).

### 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

### Входни данни

Като параметри на функцията подаваме **две цели числа**:

- Цяло число **d** в интервала **[1 ... 31]** – ден. Номерът на деня не надвишава броя дни в съответния месец (напр. 28 за февруари).
- 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).

### Изходни данни

Отпечатайте на конзолата един единствен ред, съдържащ дата след 5 дни във формат **ден.месец**. Месецът трябва да бъде двуцифрене число с водеща нула, ако е необходимо. Денят трябва да е без водеща нула.

## Насоки и подсказки

Приемаме си входа от конзолата:

```
d = Number(d);
m = Number(m);
```

За да си направим по-лесно проверките, ще си създадем една променлива, която ще съдържа **броя дни**, които има в месеца, който сме задали:

```
let daysInMonth = 31;

if (m === 2) {
    daysInMonth = 28;
}

if (m === 4 || m === 6 || m === 9 || m === 11) {
    daysInMonth = 30;
}
```

Увеличаваме **дения** с 5.

```
d += 5;
```

Проверяваме дали **денят** не е станал по-голям от **броя дни**, които има в съответния **месец**. Ако това е така, трябва да извадим дните от месеца от получения 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 създадем допълнителна променлива за месеца, към която да прибавим 0 при необходимост. Накрая принтираме дена и новата променлива за месеца:

```
let monthToPrint = m;
if (m < 10) {
    monthToPrint = '0' + monthToPrint;
}

console.log(d + '.' + monthToPrint);
```

## Тестване в Judge системата

Тествайте решението си: <https://judge.softuni.bg/Contests/Practice/Index/939#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:  $2 + 3 = 5$ .

### 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

### Входни данни

Като параметри на функцията подаваме **три цели числа**. Числата са в диапазона [1 ... 1000].

### Изходни данни

- 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" на конзолата.

### Насоки и подсказки

Приемаме си входа от конзолата.

```
a = Number(a);
b = Number(b);
c = Number(c);
```

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.log('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.log(b + ' + ' + a + ' = ' + c);
    } else {
        console.log(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.log(b + ' + ' + a + ' = ' + c);
    } else {
        console.log(a + ' + ' + b + ' = ' + c);
    }
} else if (a + c === b) {
    if (a > c) {
        console.log(c + ' + ' + a + ' = ' + b);
    } else {
```

```

    console.log(a + ' + ' + c + ' = ' + b);
}
} else if (b + c === a) {
    if (b > c) {
        console.log(c + ' + ' + b + ' = ' + a);
    } else {
        console.log(b + ' + ' + c + ' = ' + a);
    }
} else {
    console.log('No');
}

```

## Тестване в Judge системата

Тествайте решението си: <https://judge.softuni.bg/Contests/Practice/Index/939#5>.

## Задачи с единичен цикъл

Четвъртата задача на практическия изпит по “Основи на програмирането” включва единичен цикъл с проста логика 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:

- $sum1 = a_1 + a_4 + a_7 + \dots$  (сумират се числата, започвайки от първото със стъпка 3).
- $sum2 = a_2 + a_5 + a_8 + \dots$  (сумират се числата, започвайки от второто със стъпка 3).
- $sum3 = a_3 + a_6 + a_9 + \dots$  (сумират се числата, започвайки от третото със стъпка 3).

### Входни данни

Като параметър на функцията подаваме масив с големина  $n+1$  ( $0 \leq n \leq 1000$ ). Масивът ще съдържа **броя** на числата  $n$  и  $n$  цели числа в интервала  $[-1000 \dots 1000]$ :  $a_1, a_2, \dots, a_n$ .

### Изходни данни

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 (големината на входния масив) и ще декларираме начални стойности на трите суми.

```
let sum1 = 0;
let sum2 = 0;
let 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 (let i = 0; i < input.length; i++) {
    let num = input[i];

    // 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 (let i = 0; i < input.length; i++) {
    let num = input[i];

    if(i % 3 === 0) {
        sum1 += num;
    }

    if(i % 3 === 1) {
        sum2 += num;
    }
}
```

```

if(i % 3 === 2) {
    sum3 += num;
}
}

```

Накрая, ще отпечатаме резултата на конзолата в изисквания **формат**.

```

console.log('sum1 = ' + sum1);
console.log('sum2 = ' + sum2);
console.log('sum3 = ' + sum3);

```

## Тестване в Judge системата

Тествайте решението си: <https://judge.softuni.bg/Contests/Practice/Index/939#6>.

### Задача: поредица от нарастващи елементи

Дадена е редица от  $n$  числа:  $a_1, a_2, \dots, a_n$ . Да се пресметне **дълчината на най-дългата нарастваща поредица** от последователни елементи в редицата от числа.

#### Входни данни

Като параметър на функцията подаваме масив с големина  $n+1$  ( $0 \leq n \leq 1000$ ). Масивът съдържа **броя** на числата  $n$  и  $n$  цели числа в интервала  $[-1000 \dots 1000]$ :  $a_1, a_2, \dots, a_n$ .

#### Изходни данни

На конзолата трябва да се отпечата едно число – **дълчината** 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		6	
4		7		4		7	
		6		4		8	

#### Насоки и подсказки

За решението на тази задача трябва да помислим малко **по-алгоритмично**. Дадена ни е **редица от числа** и трябва да проверяваме дали всяко **следващо**, ще бъде **поголямо 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.

```
let countCurrentLongest = 0;
let countLongest = 0;
let numPrev = 0;
let num = 0;
```

Променливата **n** е **броя числа**, които ще получим от конзолата. В **countCurrentLongest** ще запазваме **броя на елементите** в нарастващата редица, която **броям в момента**. Напр. 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 элемента. Ще използваме **countLongest**, за да запазим **най-дългата** нарастваща редица. Останалите променливи са **num** - **числото**, на което се намираме **в момента**, и **numPrev** - **предишното число**, което ще сравним с **num**, за да разберем дали редицата **расте**.

Започваме да въртим числата и проверяваме дали настоящото число **a** е по-голямо от предходното **numPrev**. 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 **num** **не е по-голямо** от предходното, това означава, че започва **нова редица** и трябва да стартираме броенето от **1**. Накрая, след всички проверки, **numPrev** става **числото**, което използваме **в момента**, и започваме цикъла от начало със **следващото** въведено **num**.

Ето и примерна реализация на описания алгоритъм:

```
for (let i = 0; i < input.length; i++) {
    num = input[i];

    if (num > numPrev) {
        countCurrentLongest++;
    } else {
        countCurrentLongest = 1;
    }

    numPrev = num;
}
```

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 дължината на **най-дългата** **намерена до сега**. Целият цикъл ще изглежда така:

```

for (let i = 0; i < input.length; i++) {
    num = input[i];

    if (num > numPrev) {
        countCurrentLongest++;
    } else {
        countCurrentLongest = 1;
    }

    if (countCurrentLongest > countLongest) {
        countLongest = countCurrentLongest;
    }

    numPrev = num;
}

```

Накрая принтираме дълчината на **най-дългата** намерена редица.

```
console.log(countLongest);
```

### Тестване в Judge системата

Тествайте решението си: <https://judge.softuni.bg/Contests/Practice/Index/939#7>.

## Задачи за “чертане” на фигурки на конзолата

Петата задача на практическия изпит по “Основи на програмирането” изиска 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 алгоритично. Ето няколко примера за изпитни задачи:

### Задача: перфектен диамант

Да се напише функция, която приема като параметър цяло число **n** и чертае **перфектен диамант** с размер **n** като в примерите по-долу.

#### Входни данни

Параметър цяло число **n** в интервала [1 ... 1000].

#### Изходни данни

На конзолата трябва да се отпечата диамантът като в примерите.

#### Примерен вход и изход

Вход	Изход	Вход	Изход	Вход	Изход	Вход	Изход
2	<pre>       *      *_*       *     </pre>	3	<pre>       *      *_*     *-*_*       *_*         *       </pre>	4	<pre>       *      *_*     *-*_*_     *-*_*-       *_*         *       </pre>	5	<pre>       *      *_*     *-*_*_     *-*_*-*_     *-*_*-*-       *_*         *       </pre>

## 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 **стълки**. 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 параметрите на функцията.

```
n = Number(n);
```

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 - индекса на реда - 1** (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 звездичка**. Забележете, че започваме да броим от 0, а **не от 1**. 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 (let i = 0; i < n; i++) {
  console.log(
    ' '.repeat(n - i - 1) +
    '*'
    // TODO: Draw the rest of the line
  );
}
```

Остава да **довършим всеки ред** с нужния брой **-\*** елементи. На всеки ред трябва да добавим **i** такива **елемента** (на първия -> 0, на втория -> 1 и т.н.)

Ето и пълният код за начертаване на **горната част на диаманта**:

```

for (let i = 0; i < n; i++) {
    console.log(
        ' '.repeat(n - i - 1) +
        '*' +
        '-*'.repeat(i) +
        ' '.repeat(n - i - 1)
    );
}

```

За да изрисуваме **долната** част на диаманта, трябва да обърнем **горната** на обратно. Ще броим от **n - 2**, тъй като ако започнем от **n - 1**, 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 (let i = n - 2; i >= 0; i--) {
    console.log(
        ' '.repeat(n - i - 1) +
        '*' +
        '-*'.repeat(i) +
        ' '.repeat(n - i - 1)
    );
}

```

Остава да си сглобим **цялата програма** като първо четем параметъра на функцията, 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/939#8>.

## Задача: правоъгълник със звездички в центъра

Да се напише функция, която приема като параметър цяло число **n** и чертае правоъгълник с размер **n** с **две звездички в центъра**, като в примерите по-долу.

### Входни данни

Параметърът е цяло число **n** в интервала [2 ... 1000].

### Изходни данни

На конзолата трябва да се отпечатва правоъгълникът като в примерите.

### Примерен вход и изход

Вход	Изход	Вход	Изход	Вход	Изход	Вход	Изход
2	%%%%% %**% %%%%	3	%%%%%%%%% % % % ** % % % %%%%%%%%%	4	%%%%%%%%% % % % ** % % % %%%%%%%%%	5	%%%%%%%%% % % % % % ** % % % %%%%%%%%%

## Насоки и подсказки

Прочитаме входния параметър на функцията.

```
n = Number(n);
```

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.log('%'.repeat(2 * n));
// TODO: Draw the middle of the rectangle
console.log('%'.repeat(2 * n));
```

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.). Създаваме си променлива, която представлява броя редове, които ще има нашият правоъгълник, и 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 (ширина е **2 \* n** и вадим 2 за двета процента в края). Не забравяйте да преместите кода за **последния ред** след **цикъла**:

```
let numRows = n;
if (n % 2 === 0) numRows--;

for (let i = 0; i < numRows; i++) {
    console.log(
        '%' +
        ' '.repeat(n - 2) +
        // TODO: Place the stars
        ' '.repeat(n - 2) +
        '%'
    );
}
```

Можем да **стартираме 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 има **n - 2** празни места (**n** е половината дължина и махаме звездичката и процента), **две звезди** и отново **n - 2** празни места. Двата процента в началото и в края на реда си ги оставяме извън проверката.

```
for (let i = 0; i < numRows; i++) {
    if (i === (numRows - 1) / 2) {
        console.log(
            '%' +
            ' '.repeat(n - 2) +
            '**' +
            ' '.repeat(n - 2) +
            '%'
        );
    } else {
        console.log(
            '%' +
            ' '.repeat(n * 2 - 2) +
            '%'
        );
    }
}
```

## Тестване в Judge системата

Тествайте решението си: <https://judge.softuni.bg/Contests/Practice/Index/939#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, изискващи 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 ≤ n1 < n2 < n3 < n4 ≤ b**.

## Входни данни

Като параметри на функцията получаваме две цели числа **a** и **b** в интервала [0 ... 1000].

## Изходни данни

Изходът съдържа всички търсени четворки числа, в нарастващ ред, по една на ред.

### Примерен вход и изход

Вход	Изход
3	3 4 5 6
7	3 4 5 7 3 4 6 7 3 5 6 7 4 5 6 7

Вход	Изход
5	
7	No

Вход	Изход
10	
13	10 11 12 13

## Насоки и подсказки

Прочитаме входните параметри на функцията. Създаваме и допълнителната променлива **count**, която ще следи дали има съществуваща редица числа.

```
a = Number(a);
b = Number(b);
let count = 0;
```

Най-лесно ще решим задачата, ако логически я разделим **на части**. Ако се изисква 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 (let i = a; i <= b; i++) {
    for (let j = i + 1; j <= b; j++) {
        console.log(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 **вложените цикли** и за четири числа. Ще добавим и **увеличаване на брояча**, който инициализирахме в началото, за да знаем дали съществува такава редица.

```
for (let i = a; i <= b; i++) {
    for (let j = i + 1; j <= b; j++) {
        for (let k = j + 1; k <= b; k++) {
            for (let l = k + 1; l <= b; l++) {
                console.log(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.log('No');
```

## Тестване в Judge системата

Тествайте решението си: <https://judge.softuni.bg/Contests/Practice/Index/939#10>.

## Задача: генериране на правоъгълници

По дадено число **n** и **минимална площ m** да се генерират всички правоъгълници с цели координати в интервала  $[-n \dots n]$  с площ поне **m**. Генерираните правоъгълници да се отпечатат в следния формат:

**(left, top) (right, bottom) -> area**

Правоъгълниците се задават чрез горния си ляв и долния си десен ъгъл. В сила са следните неравенства:

- $-n \leq left < right \leq n$
- $-n \leq top < bottom \leq n$

## Входни данни

Като параметри на функцията получаваме две числа:

- Цяло число **n** в интервала  $[1 \dots 100]$  – задава минималната и максималната координата на връх.
- Цяло число **m** в интервала  $[0 \dots 50\ 000]$  – задава минималната площ на генерираните правоъгълници.

## Изходни данни

- 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

Прочитаме входните параметри на функцията. Ще създадем и един **брояч**, в който ще пазим броя на намерените правоъгълници.

```
n = Number(n);
m = Number(m);
let count = 0;
```

Изключително важно е да успеем да си представим задачата, преди да започнем да я решаваме. В нашия случай се изисква да търсим правоъгълници в координатна система. Нещото, което знаем е, че **лявата 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 **цикъл**, подобен на този 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 (let left = -n; left < n; left++) {
  for (let top = -n; top < n; top++) {
    for (let right = left + 1; right <= n; right++) {
      for (let 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 правоъгълника. 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.

```
for (let left = -n; left < n; left++) {
    for (let top = -n; top < n; top++) {
        for (let right = left + 1; right <= n; right++) {
            for (let bottom = top + 1; bottom <= n; bottom++) {
                let area =
                    Math.abs(right - left) * Math.abs(bottom - top);

                if (area >= m) {
                    console.log(
                        '(' + left + ', ' + top + ') (' +
                        right + ', ' + bottom + ') -> ' + area
                    );
                    count++;
                }
            }
        }
    }
}
```

Накрая принтираме “No”, ако не съществуват такива правоъгълници.

```
if (count === 0) console.log('No');
```

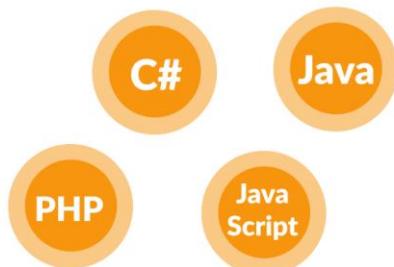
### Тестване в Judge системата

Тествайте решението си: <https://judge.softuni.bg/Contests/Practice/Index/939#11>.

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**, съдействайки 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](http://softuni.bg/apply)

# 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.

## 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 фигурка 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

На функцията се подават **4 аргумента**:

- 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	<p>Разстояние с първоначална скорост: <math>90 \text{ км/ч} * 1 \text{ час (60 мин)} = 90 \text{ км}</math></p> <p>След увеличението: <math>90 + 10\% = 99.00 \text{ км/ч} * 1.166 \text{ часа (70 мин)} = 115.50 \text{ км}</math></p> <p>След намаляването: <math>99 - 5\% = 94.05 \text{ км/ч} * 1.33 \text{ часа (80 мин)} = 125.40 \text{ км}</math></p> <p>Общо изминати: 330.9 км</p>

Вход	Изход	Обяснения
140 112 75 190	917.12	<p>Разстояние с първоначална скорост: <math>140 \text{ км/ч} * 1.86 \text{ часа (112 мин)} = 261.33 \text{ км}</math></p> <p>След увеличението: <math>140 + 10\% = 154.00 \text{ км/ч} * 1.25 \text{ часа (75 мин)} = 192.5 \text{ км}</math></p> <p>След намаляването: <math>154.00 - 5\% = 146.29 \text{ км/ч} * 3.16 \text{ часа (190 мин)} = 463.28 \text{ км}</math></p> <p>Общо изминати: 917.1166 км</p>

## 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:

- **Приемане** на входните данни.
- **Изпълнение** на основната програмна логика.
- **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 решението** на няколко логически обособени стъпки:

- **Пресмятане** на изминатото **разстояние** с първоначална скорост.
- Промяна на **скоростта** и пресмятане на изминатото **разстояние**.
- Последна промяна на **скоростта** и **пресмятане**.
- **Сумиране**.

По условие за **входни данни** ще ни бъдат подадени **четири** аргумента на функцията, които трябва да **преобразуваме** в числа, за да можем да извършим

необходимите пресмятания. Преобразуването ще направим с помощта на **Number(...)** конструктора:

```
let initialSpeed = Number(args[0]);
// let firstInterval = TODO
// let secondInterval = TODO
// let thirdInterval = TODO
```

По този начин успяхме да се справим успешно с **първата подзадача** - приемане на входните данни.

Първоначално **запазваме** една **променлива**, която ще използваме многократно. 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:

```
let minutesPerHour = 60;
```



Избягването на **повтарящ 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** подаденото ни **време 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 получаваме 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.

```
let firstIntervalHours = firstInterval / minutesPerHour;
let firstDistance = initialSpeed * firstIntervalHours;

let speedAfterIncrease = initialSpeed + ((initialSpeed * 10) / 100);
let secondIntervalHours = secondInterval / minutesPerHour;
let secondDistance = speedAfterIncrease * secondIntervalHours;
// TODO: Calculate thirdDistance
```

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 направим това по следния начин:

```
let finalDistance =
    firstDistance + secondDistance + thirdDistance;
console.log(finalDistance.toFixed(2));
```

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/940#0>.

## Задача: смяна на плочки

Хараламби има събрани пари, с които иска да **смени плочките** на пода в банята. Като подът е правоъгълник, а плоцките са триъгълни. Напишете програма, която да пресмята дали събранныте пари ще му стигнат. Подават се широчината и дължината на пода, както и едната страна на триъгълника с височината към нея. Трябва да пресметнете колко плочки са нужни, за да се покрие пода. Броят на плоцките трябва да се закръгли към по-високо цяло число и да се прибавят още 5 броя за фирма. В допълнение ни се подават още – цената на плошка и сумата за работата на майстор.

## Входни данни

Като параметри на функцията подаваме 7 числа:

- Събрани пари.
- Широчината на пода.
- Дължината на пода.
- Страната на триъгълника.
- Височината на триъгълника.
- Цена на една плошка.
- 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:

- "{Оставащите 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 $\rightarrow 5.55 * 8.95 = 49.67249$
5.55		Площта на плочка $\rightarrow 0.9 * 0.85 / 2 = 0.3825$
8.95	You'll need 1209.65 lv more.	Необходими плочки $\rightarrow 49.67249 / 0.3825 = 129.86\dots$ $= 130 + 5$ фира $= 135$
0.90		Обща сума $\rightarrow 135 * 13.99 + 321$ (майстор) $= 2209.65$
0.85		$2209.65 > 1000 \rightarrow$ не достигат 1209.65 лева
13.99		
321		

Вход	Изход	Обяснения
500		Площ на пода $\rightarrow 3 * 2.5 = 7.5$
3		Площта на плочка $\rightarrow 0.5 * 0.7 / 2 = 0.175$
2.5		Необходими плочки $\rightarrow 7.5 / 0.175 = 42.857\dots = 43 + 5$
0.5		фира $= 48$
0.7		Обща сума $\rightarrow 48 * 7.8 + 100$ (майстор) $= 474.4$
7.80	25.60 lv left.	$474.4 < 500 \rightarrow$ остават 25.60 лева
100		

## Насоки и подсказки

Следващата задача изисква от нашата функция да приема повече входни данни и извърши по-голям брой изчисления, въпреки че решението е **идентично**. Приемато на данните от потребителя извършваме по добре **познатия ни** вече начин.

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 покриването 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.ceil(...)**.

До крайния резултат можем да стигнем, като **пресметнем общата сума**, която е необходима, за да бъде покрит целият 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 една плочка**. Дали сумата, с която разполагаме, ще бъде достатъчна, разбираме като сравним събраните до момента пари (от входните данни) и общите разходи:

```
if (budget >= totalCost){
    // TODO: Print message
}
else {
    // TODO: Print message
}
```

## Тестване в Judge системата

Тествайте решението си: <https://judge.softuni.bg/Contests/Practice/Index/940#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 осъществяването за празничен ден!

Цената за аранжиране на букета винаги е **2 лв.** Напишете програма, която изчислява **цената за един букет**.

## Входни данни

Функцията приема 5 аргумента:

- **Броят** на закупените **хризантеми** – цяло число в интервала [0 ... 200].
- **Броят** на закупените **рози** – цяло число в интервала [0 ... 200].
- **Броят** на закупените **лалета** – цяло число в интервала [0 ... 200].
- **Сезонът** – [Spring, Summer, Autumn, Winter].
- **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: <b>няма намаление</b>
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
let roseAutumnWinterPrice = 4.5;
```

```
let roseSpringSummerPrice = 4.1;
let tulipAutumnWinterPrice = 4.15;
let tulipSpringSummerPrice = 2.5;
let chrysantemumAutumnWinterPrice = 3.75;
let chrysantemumSpringSummerPrice = 2;
let arrangePrice = 2;
```

Правим същото и за останалите вече дефинирани стойности:

```
// Price increases
let priceIncreasePercentage = 15;

// Price decreases
let tulipPriceDecreasePercentage = 5;
let rosePriceDecreasePercentage = 10;
let totalPriceDecreasePercentage = 20;

// Price decrease thresholds
let tulipPriceDecreaseThreshold = 7;
let rosePriceDecreaseThreshold = 10;
let totalPriceDecreaseThreshold = 20;
```

Следващата ни подзадача е да обработим правилно входните данни на функцията. Подхождаме по добре познатия ни вече начин за преобразуването им в числен тип данни:

```
let chrysantemumsPurchased = Number(args[0]);
// let rosesPurchased = ....
```

Нека помислим кой е най-подходящият начин да структурираме нашата програмна логика. От условието става ясно, че пътят на програмата се разделя основно на две части: пролет / лято и есен / зима. Разделението ще направим с условна конструкция **if-else**, като преди това заделяме променливи за цените на отделните цветя, както и за крайния резултат:

```
if (season === "Winter" || season === "Autumn") {
    rosesPrice = rosesPurchased * roseAutumnWinterPrice;
    chrysantemumsPrice = chrysantemumsPurchased *
        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/940#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 **изпечата 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

Програмата прочита **поредица от числа** (аргумента):

- На първия ред (аргумент) – **броят на студентите явили се на изпит** – цяло число в интервала [1 ... 1000].
- За **всеки един студент** на отделен ред (аргумент) – **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}".

Резултатите трябва да са **форматирани до втория символ** след десетичния знак.

#### Примерен вход и изход

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	

Вход	Изход	Обяснения
10		
3.00		
2.99		5 и повече – <b>трима</b> = 30% от 10
5.68	Top students: 30.00%	Между 4.00 и 4.99 – <b>трима</b> = 30% от 10
3.01	Between 4.00 and 4.99: 30.00%	Между 3.00 и 3.99 – <b>двама</b> = 20% от 10
4	Between 3.00 and 3.99: 20.00%	Под 3 – <b>двама</b> = 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		

## 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** ще приемем **броя** на студентите. За да обработим самите оценки, ще използваме **for** цикъл. Всяка итерация на цикъла ще прочита и обработва по една оценка:

```
let numberOfRowsStudents = Number(args[0]);
for (var i = 1; i < numberOfRowsStudents; 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:

```
let numberFailedStudents = 0;
let numberAverageStudents = 0;
let numberGoodStudents = 0;
let numberExcellentStudents = 0;
let totalResult = 0;
```

Завъртаме цикъла и в него **декларираме още една** променлива, в която ще запазваме **текущата** въведена оценка. Променливата ще е от тип **Number** и на всяка итерация ще 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 (var i = 1; i < number0fStudents; i++) {
    let grade = Number(args[i]);
    totalResult += grade;
    if (grade < 3) {
        number0fFailedStudents++;
    } 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.

**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/940#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 по-долу.

### Примерен вход и изход

Вход	Изход	Вход	Изход
4	<pre> ...../ \... .....\ /... ....***... ....*-*-*... ....*-*-*... ....*-*-*... ....*-----*... ...*-----*... .*-----*... *-----*... *****... *.*.*.*.*... *****... </pre>	7	<pre> ...../ \... .....\ /... ....***... ....*-*-*... ....*-*-*... ....*-*-*... ....*-----*... ...*-----*... .*-----*... *-----*... *****... *-----*... *-----*... *****... </pre>

## Входни данни

На функцията се подава само един аргумент - цяло число n в интервала [3 ... 100].

## Изходни данни

Да се отпечата на конзолата **коледна шапка**, точно както в примерите.

## Насоки и подсказки

При задачите за **чертане** на 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**.

The image shows a decorative horizontal border. The top part is a continuous line of asterisks (\*). The bottom part consists of a repeating pattern of an asterisk (\*) followed by a dot (.).

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** част от фигурката ще използваме **цикъл**. Размерът на цикъла ще бъде от **0 до широчината**, 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 всеки** начертан ред **намаляваме** броя на точките с **1**, а **ти retata увеличим** с **1**. Нека не забравяме да начертаем и по една **звездишка** 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/940#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 на програмата съдържа **точно 3 реда** (аргумента):

- Малка буква от английската азбука за начало на интервала – от 'a' до 'z'.
- Малка буква от английската азбука за край на интервала – от **първата буква** до 'z'.
- Малка буква от английската азбука – от 'a' до 'z' – като комбинациите, съдържащи 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 bcccaa cab cac cba cbb cbc cca ccb ccc Комбинациите, <b>съдържащи 'b'</b> , не са валидни. Остават <b>8</b> валидни комбинации.

Вход	Изход
a c z	aaa aab aac aba abb abc aca acb acc baa bab bac bba bbb bbc bca bcb bcccaa cab cac cba cbb cbc cca ccb ccc 27

Вход	Изход
f k h	ffff ffg ffi ffj fff gfg 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 ggj ggk gif gig giij gik gif gig gjij gjk gjk gkf gkg gki gkj gkk iff ifg ifi ifk ifg ifg igg igi igj igk iif iig iii iij iik ijf ijg iji ijj ijk ikf ikg iki ikj ikk jff jfg jfi jff jfk jgf jgg jgi jgj jgk jif jig jii jjj jik jjf jjg jjj jjk jkf jkg jki jkj jkk kff kfg kfi kfj kfk kgf kgg kgi kgj kgk kif kig kii kij kik kfj kjf kji kjj kjk kkf kkg kki kkj kkk 125

## Насоки и подсказки

За последната задача имаме по условие входни данни от **3 аргумента**, които са представени от по един символ от **ASCII таблицата** (<http://www.asciitable.com/>). Бихме могли да използваме вече дефинирания метод в езика JavaScript, **.charCodeAt()**, чрез който ще получим ASCII кода на подадения символ:

```
let startLetter = args[0].charCodeAt();
// 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**, с който да преминем през **всички символи** и открием тези, които са **различни от буквата**, която трябва да пропуснем. В JavaScript можем да обходим всички символи от 'a' до 'z' ето така:

```
let result = "";
for (let i = 'a'.charCodeAt(); i <= 'z'.charCodeAt(); i++) {
    result += String.fromCharCode(i) + " ";
}

console.log(result);
```

Методът **String.fromCharCode(...)** ще конвертира подадения ASCII код в символ. Резултатът от изпълнението на горния код 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. Там най-често 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 ще имаме три вложени цикъла, като в тялото на **последния** ще добавяме символите към крайния резултат:

```
for (char i = startLetter; i <= endLetter; i++)
{
    if (i != exceptLetter)
    {
        // TODO
    }
}
```

Нека не забравяме, че се изисква 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. Тази подзадача оставяме на читателя.

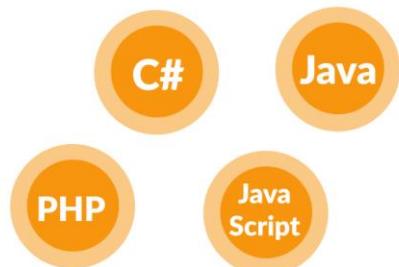
## Тестване в Judge системата

Тествайте решението си: <https://judge.softuni.bg/Contests/Practice/Index/940#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



**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](http://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 развиwanе 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 Фиbonачи), 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 в редицата 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.

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.

45	...	...	...	...	...	55
...	17	19	21	23	...	...
...	15	5	7	...	...	...
...	13	11	9	...	...	...
37	...	...	...	29	...	65
...	...	...	...	...	...	...

### Входни данни

Като параметри на функцията подаваме **5 цели числа**.

- Първите **три параметъра** представляват **първите три числа** в редицата на Трибоначи, положителни ненулеви числа, сортирани в нарастващ ред.
- Следващите **два параметъра**, представляват **първото число и стъпката** за всяка клетка на матрицата за спиралата от числа. Числата, описващи спиралата, са положителни ненулеви 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: 0.3 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 обработим входа. Входните данни се състоят от **5 цели числа**: 3 за редицата на Трибоначи и 2 за числовата спирала:

```
let tribonacciFirst = Number(args[0]);
let tribonacciSecond = Number(args[1]);
let tribonacciThird = Number(args[2]);
```

```
let spiralCurrent = Number(args[3]);
let spiralStep = Number(args[4]);
```

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 числото, 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:

```
let tribonacciNumbers = [tribonacciFirst,
    |           |           |
    |           |           | tribonacciSecond,
    |           |           | tribonacciThird];

let tribonacciCurrent = tribonacciThird;

while (tribonacciCurrent < 1000000) {
    tribonacciCurrent = tribonacciFirst +
        tribonacciSecond + tribonacciThird;

    tribonacciNumbers.push(tribonacciCurrent);

    tribonacciFirst = tribonacciSecond;
    tribonacciSecond = tribonacciThird;
    tribonacciThird = tribonacciCurrent;
}
```

## Генериране на числовая 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 картинаката 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 проверка (`spiralCount % 2 == 0`). Целият код от генерирането на спиралата в **масив** е даден по-долу:

```
let spiralNumbers = [spiralCurrent];
let spiralCount = 0, spiralStepMul = 1;
while (spiralCurrent < 1000000) {
    spiralCurrent += spiralStep * spiralStepMul;
    spiralNumbers.push(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** (Binary Search), тъй като вторият масив се генерира сортиран, т.е. отговаря на изискването за прилагането на този тип търсене. Кодът за намиране на нашето решение ще изглежда така:

```
let found = false;
for (let i = 0; i < tribonacciNumbers.length; i++) {
    for (let j = 0; j < spiralNumbers.length; j++) {
        if (tribonacciNumbers[i] == spiralNumbers[j] &&
            tribonacciNumbers[i] <= 1000000) {
            console.log(tribonacciNumbers[i]);
            found = true;
            break;
    }
}
```

```

}

if (found) {
    break;
}

if (!found) {
    console.log("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/941#0>.

## Задача: магически дати

Дадена е **дата** във формат "дд-мм-гггг", напр. 17-04-2018. Изчисляваме **теглото на тази дата**, като вземем всичките ѝ цифри, умножим всяка цифра с останалите след нея и накрая съберем всички получени резултати. В нашия случай имаме 8 цифри: 17032007, така че теглото е  **$1*7 + 1*0 + 1*3 + 1*2 + 1*0 + 1*0 + 1*7$**

$$\begin{aligned}
 & + 7*0 + 7*3 + 7*2 + 7*0 + 7*0 + 7*7 + 0*3 + 0*2 + 0*0 + 0*0 + 0*7 + 3*2 \\
 & + 3*0 + 3*0 + 3*7 + 2*0 + 2*0 + 2*7 + 0*0 + 0*7 + 0*7 = 144.
 \end{aligned}$$

Нашата задача е да напишем програма, която намира всички **магически дати** - дати между две определени години (включително), отговарящи на дадено във входните данни тегло. Датите трябва да бъдат принтирани в нарастващ ред (по дата) във формат "**дд-мм-гггг**". Ще използваме само валидните дати в традиционния календар (високосните години имат 29 дни през февруари).

## Примерен вход и изход

Вход	Изход
2007	17-03-2007
2007	13-07-2007
144	31-07-2007

Вход	Изход
2003	
2004	No
1500	

Вход	Изход
2011	01-01-2011
2012	10-01-2011
14	01-10-2011
	10-10-2011

Вход	Изход
	09-01-2013
	17-01-2013
	23-03-2013
	11-07-2013
	01-09-2013
2012	10-09-2013
2014	09-10-2013
80	17-10-2013
	07-11-2013
	24-11-2013
	14-12-2013
	23-11-2014
	13-12-2014
	31-12-2014

## Входни данни

Входните данни съдържат 3 цели числа:

- Първото цяло число: **начална година**.
- Второто цяло число: **крайна година**.
- Третото цяло число: **търсеното тегло** за датите.

Входните данни винаги ще бъдат валидни и винаги ще са в описания формат. Няма нужда да се проверяват.

## Изходни данни

Резултатът трябва да бъде принтиран на конзолата, като последователни дати във формат "**дд-мм-гггг**", подредени по дата в нарастващ ред. Всеки низ трябва да

е на отделен ред. В случай, че няма съществуващи магически дати, да се принтира "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.

## Насоки и подсказки

Започваме от входните данни. В случая имаме 3 цели числа:

```
let firstYear = Number(args[0]);
let secondYear = Number(args[1]);
let numberToSearchFor = Number(args[2]);
```

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, която ни дава **Date** обектът в JavaScript. 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 получили като параметър, а месеца и деня трябва да са съответно януари и 1-ви. При JavaScript "конструкторът" на **Date** приема като първи аргумент годината, като втори аргумент месеца (0 е януари, 11 е декември) и като трети аргумент деня от месеца:

```
let date = new Date(firstYear, 0, 1);
```

След като имаме началната дата, искаме да направим **цикъл**, който се изпълнява, **докато не превишам 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 метод от **Date - setDate(...)**, чрез който ще добавяме по един ден към текущата дата, която пък ще вземем с **getDate()**. Методът ще се грижи вместо нас кога трябва да прескочи в следващия месец, колко дни има даден месец и всичко около високосните години:

```
date.setDate(date.getDate() + 1);
```

В JavaScript трябва да използваме метода `getFullYear()` за взимането на годината във формат подобен на дадения във входните данни. Ако използваме метода `getYear()`, ще получим броя години минали от 1900 до търсената дата, което не ни върши работа в текущата задача. В крайна сметка нашият цикъл ще изглежда по следния начин:

```
while (date.getFullYear() <= secondYear) {

    // TODO: Check date for numberToSearchFor

    date.setDate(date.getDate() + 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.

## 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 методите `getDate()`, `getMonth()` и `getFullYear()`, можем да извлечем всяка цифра. Трябва да внимаваме с `getMonth()`, защото по аналогия с конструктора, този метод връща число между 0 (януари) и 11 (декември) и при него е необходимо добавяне на **+1**, за да получим месеца в интервал **[1-12]**. Друго, за което трябва да внимаваме, е целочисленото деление на 10 (**/ 10**), което няма да е целочислено тук и затова след всяко целочислено деление ще закръглеме изрично до най-малкото цяло число, чрез метода `Math.floor(...)`:

```
let d1 = Math.floor(date.getDate() / 10); // First day digit
let d2 = date.getDate() % 10; // Second day digit

let d3 = Math.floor((date.getMonth() + 1) / 10); // First month digit
let d4 = (date.getMonth() + 1) % 10; // Second month digit

let d5 = Math.floor(date.getFullYear() / 1000); // First year digit
let d6 = Math.floor(date.getFullYear() / 100) % 10; // Second year digit
let d7 = Math.floor(date.getFullYear() / 10) % 10; // Third year digit
let d8 = date.getFullYear() % 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 + ... + d1 \* d8**, може да съкратим този израз до **d1 \* (d2 + d3 + ... + d8)**, следвайки математическите правила за групиране, когато имаме умножение и събиране. Прилагайки същото опростяване за останалите умножения, получаваме следната формула:

```
weight = d1 * (d2 + d3 + d4 + d5 + d6 + d7 + d8) +
         d2 * (d3 + d4 + d5 + d6 + d7 + d8) +
         d3 * (d4 + d5 + d6 + d7 + d8) +
         d4 * (d5 + d6 + d7 + d8) +
         d5 * (d6 + d7 + d8) +
         d6 * (d7 + d8) +
         d7 * d8;
```

## Отпечатване на изхода

След като имаме пресметнато 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. За щастие имаме вече всяка една от цифрите, които трябва да отпечатаме, а именно **d1** до **d8**. Тук трябва да се внимава с типовете данни. Тъй като конкатенацията на низове и операцията събиране на числа се извършват с един и същ оператор, трябва да конвертираме цифрите към низове или да започнем конкатенацията от празен символен низ:

```
if (weight == numberToSearchFor) {
    console.log("+" + d1 + d2 + "-" + d3 + d4 + "-" + d5 + d6 + d7 + d8);
    found = true;
}
```

**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.log("No");
}
```

## Тестване в Judge системата

Тествайте решението си: <https://judge.softuni.bg/Contests/Practice/Index/941#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 **c1, c2, ..., cn** е изчислено, като се премахват всички букви, които се повтарят (от дясно наляво), и след това се пресметне формулата:

$$\text{weight}(c_1, c_2, \dots, c_n) = 1 * \text{weight}(c_1) + 2 * \text{weight}(c_2) + \dots + n * \text{weight}(c_n)$$

Например, теглото на **bcd dc** се изчислява по следния начин:

Първо премахваме повтарящите 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:  $\text{weight}("cadae") = \text{weight}("cade") = 1 * 47 + 2 * 5 + 3 * 7 + 4 * (-32) = -50$ .

## Входни данни

Входните данни съдържат две цели числа:

- Числото за **начало**.
- Числото за **край**.

Входните данни винаги ще бъдат валидни и винаги ще са в описания формат. Няма нужда да се проверяват.

## 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, да се принтира "No".

## Ограничения

- Числата за начало и край да бъдат цели числа 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 42	bcead bdcea	weight("bcead") = 41 weight("bdcea") = 40		
			300 400	No

Вход	Изход	Вход	Изход
-1 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	200 300	baadc babdc badac badbc badca badcb badcc badcd baddc bbadc bbdac bdaac bdabc bdaca bdacb bdacc bdacd bdadc bdbac bddac beadc bedac eabdc ebadc ebdac edbac

## Насоки и подсказки

Като всяка задача, започваме решението с обработване на входните данни:

```
let firstNumber = Number(args[0]);
let secondNumber = Number(args[1]);
```

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 заданияния 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 искаме да отговаря на

един символ. За да генерираме всички комбинации с дължина 2 използвайки 5 символа (т.е. "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 (let i1 = 0; i1 < 5; i1++) {  
    for (let i2 = 0; i2 < 5; i2++) {  
        for (let i3 = 0; i3 < 5; i3++) {  
            for (let i4 = 0; i4 < 5; i4++) {  
                for (let 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:

```
let 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:

```
let 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 **String.fromCharCode('a'.charCodeAt(0) + 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:

```
let word = pattern[i1];
```

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:

```
let word = pattern[i1];
```

```
if (word.indexOf(pattern[i2]) == -1) word += pattern[i2];
if (word.indexOf(pattern[i3]) == -1) word += pattern[i3];
if (word.indexOf(pattern[i4]) == -1) word += pattern[i4];
if (word.indexOf(pattern[i5]) == -1) word += pattern[i5];
```

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, ето така:

```

let 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.e. индекс 0 е позиция 1, индекс 1 е позиция 2 и т.н., ще добавим 1 към индексите.

```
weight += multiplier * (i + 1);
```

Всички получени междуинни 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 уникални букви. Тя ни бе необходима само за пресмятане на тежестта!

Думите са **разделени с интервал** и ще ги натрупваме в междуинна 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 == "") {  
    console.log("No");  
} else {  
    console.log(result.trim());  
}
```

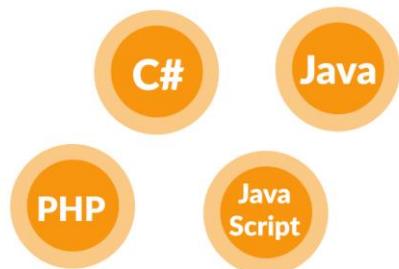
## Тестване в Judge системата

Тествайте решението си: <https://judge.softuni.bg/Contests/Practice/Index/941#2>.

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](http://softuni.bg/apply)

# Глава 9.2. Задачи за шампиони – част II

В тази глава ще разгледаме още три 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 JavaScript** (функции, 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.

Първият аргумент на функцията ще бъде **сумата**, която Лина има **преди** да започне да пазарува. Вторият аргумент ще бъде списък от команди(стрингове), които Лина ще изпълни. При получаване на командалата "**mall.Enter**", Лина влиза в мола и започва да пазарува, докато не получи командалата "**mall.Exit**". След като Лина започне да пазарува, **всеки следващ елемент** от масива, ще представлява **действия**, които Лина **изпълнява**. Всеки **символ** в стринга представлява **покупка или друго действие**. Стринговите 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 се подават в два аргумента. **Първият** ще бъде **сумата**, която Лина **има преди да започне да пазарува**. Вторият аргумент ще бъде масив от команди, който се обработва последователно. Когато получите команда **"`mall.Enter`"**, всеки следващ елемент ще бъде стринг, съдържащи **информация относно покупките / действията**, които Лина иска да направи. В масива ще има команди, които трябва да се изпълнят, 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 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 mall.Enter d mall.Exit	1 purchases. Money left: 80.00 lv.	'd' има ASCII код 100. 'd' е малка буква и за това Лина получава 70% отстъпка. 100% – 70% = 30. 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:

- В **първия аргумент** имаме **всички пари**, с които Лина ще разполага за пазаруването.
- Във **втория - масив** ще имаме поредица от **команди**.

Имайки директно парите, с които Лина разполага можем да пристъпим към обработка на командите, които сме получили. При тях, обаче, има детайл, с който трябва да внимаваме. Условието гласи следното:

Вторият аргумент ще бъде масив от команди, който се обработва последователно. Когато получите командата "**mall.Enter**", всеки следващ елемент ще бъде стринг, съдържащи **информация** относно **покупките / действията**, които Лина иска да направи.

Тук е моментът, в който трябва да съобразим, че в **масива** трябва да започнем **да обработваме команди**, но **едва след като получим** командата "**mall.Enter**". 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 команда "**mall.Enter**":

```
while (command !== 'mall.Enter') {
    i++;
    command = command[i];
    // command = command[+i];
}
```

Може да замените този **while** с **for** цикъл използвайки само условието и стъпката на **for**.

Тук е мястото да отбележим, че извикването на **i++** след края на цикъла се използва за **преминаване към първата команда** за обработка, защото в края на цикъла **command[i]** сочи точно към "**mall.Enter**", което не трябва да се обработва като действие в мола.

### Алгоритъм за решаване на задачата

Алгоритъмът за решаването на самата задача е праволинеен - продължаваме да обработваме команди, докато не бъде подадена команда "**mall.Exit**". През това време разглеждаме 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 са от тип **string**, то най-лесният начин да достъпим всеки знак в тях е чрез **for цикъл**.

Ето как би изглеждало използване на два такива цикъла:

```
for (; commands[i] !== 'mall.Exit'; i++) {
    command = commands[i];

    for (let action of command) {
        // TODO: Обработване на команди
    }
}
```

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 метода **`str.toUpperCase()`** и да сравним дали символът е същия, като този, който ще получим от **`str.toUpperCase()`**.

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') {
    let price = action.charCodeAt() * 0.5;

    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

В края на задачата трябва да **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. На база променливата за брой покупки можем да определим кой вариант на изхода ни трябва. 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/942#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. Израза **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 аргумент. Той съдържа **математическият израз за пресмятане**. Аргументът 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 – резултата 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

Обикновено, първо ще прочетем и обработим входа, след това решаваме задачата и накрая отпечатаме резултата, форматиран, според условието. В случая входът се състои от 1 аргумент, който няма нужда да бъде обработван допълнително. Затова директно преминаваме към решаването на задачата.

## Алгоритъм за решаване на задачата

За целите на нашата задача ще имаме нужда от няколко променливи:

- Една променлива, в която ще пазим **текущия резултат**.
- Една променлива, в която ще пазим до **кой индекс сме стигнали** в обхождането на нашия израз.
- Една променлива, в която ще пази **текущия символ**, който обработваме.
- И последна променлива, в която ще пазим **текущия оператор** от нашия израз.

```
let result = 0;
let index = 0;
let symbol = expression[index];
let expressionOperator = '+';
```

След като вече имаме началните 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 обработваме символи, докато не срещнем **=**. Следва точното изписване на **while цикъл**.

```
while (symbol != '=') {
    // TODO: Обработване на символите
}
```

Следващата стъпка е обработването на нашата **symbol** променлива. За нея имаме 3 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 кода на нашия символ] - [ASCII кода на символа 0] = [цифрата, която репрезентира символа]**. Ако резултатът от тази проверка е между 0 и 9, то тогава нашият символ наистина е **число**. (Алтернативно можем да използваме директно символите **'0'** и **'9'** или техните **ASCII кодове**.)
- Ако символът е **оператор**, т.е. е **+**, **-**, **\*** или **/**.

```

if (symbol == '(') {
    // TODO: Обработване на вложен израз
}
else if (0 <= symbol - '0' && symbol - '0' <= 9) {
    // TODO: Обработване на число
}
else if (symbol == '+' ||
         symbol == '-' ||
         symbol == '/' ||
         symbol == '*') {
    // TODO: Обработване на оператор
}

```

Нека разгледаме действията, които трябва да извършим при съответните случаи, които дефинирахме:

- Ако нашият символ е **оператор**, то тогава 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 кода на нашия символ] - [ASCII кода на символа0] = [цифрата, която репрезентира символа]**)

```

else if (0 <= symbol - '0' && symbol - '0' <= 9) {
    switch (expressionOperator) {
        case '+':
            result += symbol - '0';
            break;
        // TODO: Дописване на липсващите оператори
    }
}
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 == '(') {
    let innerResult = 0;
    let innerOperator = '+';
    index++;
    symbol = expression[index]; // ++index
```

След това, за **пресмятане стойността на подизраза** използваме същите методи, които използвахме за пресмятане на главния израз - използваме **while цикъл**, за **да обработваме символи** (докато не срецнем 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;
    // TODO: Дописване на липсващите оператори
}
```

## 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/942#1>.

## Задача: бикове и крави

Всички знаем играта „Бикове и крави“ ([http://en.wikipedia.org/wiki/Bulls\\_and\\_cows](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 състоят от 3 аргумента:

- Първият съдържа **секретното число**.
- Вторият съдържа **броя бикове**.
- Третият 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]**. [TODO: Цифрите е хубаво/трябва да са уникални. Ако имам тайно число 2132 и предположение 8762 имам 1 бик и 1 крава за 1 число. ]

- Броят на **кравите и биковете** винаги ще е в интервала [0...9]. [TODO: Има ли смисъл да имаме крави и бикове от 0...9? Кога ще имам 5 бика и 7 крави? Входът трябва да е валиден.]
- Позволено време за изпълнение: **0.15 секунди**.
- Позволена памет: **16 MB**.

## Примерен вход и изход

Вход	Изход
2228	1222 2122 2212 2232 2242 2252 2262 2272 2281 2283 2284 2285
2	2286 2287 2289 2292 2322 2422 2522 2622 2722 2821 2823 2824
1	2825 2826 2827 2829 2922 3222 4222 5222 6222 7222 8221 8223 8224 8225 8226 8227 8229 9222

Вход	Изход
1234	1134 1214 1224 1231 1232 1233 1235 1236 1237 1238 1239 1244
3	1254 1264 1274 1284 1294 1334 1434 1534 1634 1734 1834 1934
0	2234 3234 4234 5234 6234 7234 8234 9234

## Насоки и подсказки

Тъй като входът пристига директно, като аргументи на функцията ни остават следните стъпки от решението:

- 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.

## 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:

```
let 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.log('No');
}
```

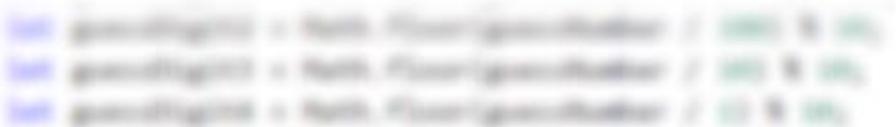
Нека започнем да размишляваме над нашия проблем. Това, което трябва да направим, е да **анализираме всички числа от 1111 до 9999** без тези, които съдържат в себе си нули (напр. **9011, 3401** и т.н. са невалидни). Какъв е най-лесният начин за **генериране** на всички тези **числа**? С **вложени цикъла**. 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.

Алтернативен подход е да обиколите с един цикъл числата от 1111 до 9999 и да прескачате всички числа с '0' в себе си, но това би променило малко решението, което ще изпълним по-долу.

```
for (let digit1 = 1; digit1 <= 9; digit1++) {
    for (let digit2 = 1; digit2 <= 9; digit2++) {
        for (let digit3 = 1; digit3 <= 9; digit3++) {
            for (let 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**.

```
let guessDigit1 = Math.floor(guessNumber / 1000) % 10;
```



Как? При деление получаваме дробно число. Или трябва да премахнем дробната част преди да разделим модулно на 10 чрез **Math.floor(...)** или чрез кастване към цяло число чрез **parseInt(...)**. В примера по-горе премахваме дробната част.

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.

```
let digitToCheck1 = digit1;
let digitToCheck2 = digit2;
let digitToCheck3 = digit3;
let digitToCheck4 = digit4;

let currentBulls = 0;
let 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 четвърта 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. Можем да обединим всички проверки в едно условие, знаейки, че и в трите случая имаме крава, но няма да знаем коя цифра да премахнем, затова ги изписваме едно по едно:

```
// 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;
}
```

```

    if (currentBulls == targetBulls && currentCows == targetCows) {
        if (solutionFound) {
            console.log(" ");
        }

        console.log(digit1 + '' + digit2 + '' + digit3 + '' + digit4);
        solutionFound = true;
    }
}

```

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 изхода

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.log(" ");

        console.log(digit1 + '' + digit2 + '' + digit3 + '' + digit4);
        solutionFound = true;
    }
}

```

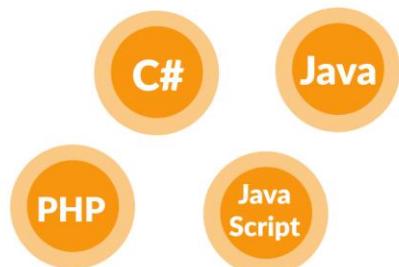
### Тестване в Judge системата

Тествайте решението си: <https://judge.softuni.bg/Contests/Practice/Index/942#2>.

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](http://softuni.bg/apply)

# Глава 10. ФУНКЦИИ

JavaScript е известен като функционален език за програмиране. Самото наименование подсказва, че **функциите** са изключително важна част от езика.

В настоящата глава ще се запознаем с **функциите** 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 от функция**, както и как да използваме тази връщана стойност. Накрая на главата, ще разгледаме **утвърдените практики** при използване на функциите.

## Какво е "функция"?

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 се нарича **функция**. Точно това представляват **функциите** – **парчета код, които са именувани** от нас по определен начин и които могат да бъдат **извикани** толкова пъти, колкото имаме нужда и ще бъдат изпълнени съответния брой пъти.

Една **функция** може да бъде **извикана** 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 функции

Простите функции 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 **проста функция**:

```
function printHeader() {  
    console.log("-----");  
}
```

Тази **функция** има задачата да отпечата заглавие, което представлява поредица от символа **-**. Поради тази причина името ѝ е **printHeader**. Кръглите скоби (**(** и **)** **винаги** следват името, независимо как сме именували функцията. По-късно ще разгледаме как трябва да именуваме функциите, с които работим, а за момента ще отбележим само, че е важно **името на функцията да описва действието**, което тя извършва.

Тялото на функцията съдържа **програмния код**, който се намира между къдравите скоби **{ и }**. Между тях поставяме кода, който решава проблема, описан от името на функцията.

## Защо да използваме функции?

До тук установихме, че функциите 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 функциите **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 отделена функция**.

Функциите ни предоставят **възможността** да използваме даден **код няколко пъти**. С решаването на все повече и повече задачи ще установите, че използването на вече деклариирани функции спестява много време и усилия.

## Деклариране на функции

В езика **JavaScript** можем да **декларираме** функции буквально навсякъде, по същият начин, по който можем да декларираме и променливи навсякъде. Декларирането представлява **регистрирането на функцията** в програмата, за да бъде разпозната в останалата част от нея.

JavaScript не е **силно типизиран** език (strongly typed). Затова и когато **декларираме функция** тя няма тип (string, number, array и т.н.), каквъто имат методите и функциите в другите езици за програмиране.

Има 2 основни начина, по които могат да се декларирате функции в JavaScript - **декларативно - function declaration** и **експресивно - function expression**.

### Декларативно (function declaration)

Със следващия пример ще разгледаме задължителните елементи в декларацията на една функция **декларативно (function declaration)**.

```
function getSquare(n) {
  return n * n;
}
```

- Ключовата думичка **function**. Започваме с използването на **ключовата думичка function**, чрез която заявяваме, че предстои декларация на функция.

Наричаме я **ключова**, защото тя е запазена в езика **JavaScript** или с други думи казано - не можем да имаме променлива, която да именуваме по този начин.

- **Име на функцията.** Името на функцията е **определеното от нас**, като не забравяме, че трябва да **описва задачата**, която се изпълнявана от кода в тялото на функцията. В примера името е **getSquare**, което ни указва, че задачата на тази функция 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 функцията 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 **n**.
- **Тяло на функцията.** Декларира се между скобите **{ и }**, които изписваме веднага след затварящата **)**. В **тялото на функцията** описваме чрез код всички операции, които искаме функцията да извърши. В тялото на функцията описваме **алгоритъма**, по който функцията решава даден проблем. Реализираме **логиката** на функцията. В показания пример изчисляваме лицето на квадрат, а именно  **$n * n$** .

При деклариране на функции е важно да спазваме **последователността** на основните елементи - първо **ключовата думичка function**, след това **име на функцията, списък от параметри**, ограден с кръгли скоби **()** и накрая **тяло на функцията**, оградено с фигурни скоби **{}**.

## Експресивно (function expression)

Със следващия пример ще разгледаме задължителните елементи в декларацията на една функция **експресивно (function expression)**. То доста наподобява **декларативното**, което вече разгледахме и може да се каже, че е **комбинация от деклариране на променлива и деклариране на функция декларативно (function declaration)**.

```
let getSquare = function getSquareFunc(n) {
    return n * n;
}
```

- **Ключовата думичка let.** Започваме с използването на **ключовата думичка let**, чрез която заявяваме, че предстои декларация на променлива.
- **Име на променливата.** Името на променливата е **определеното от нас**. В примера името е **getSquare**, което ни указва, че задачата на тази функция е да изчисли лицето на квадрат.
- **Декларация на функция.** Използвайки същата структура, която вече научихме при **function declaration** - първо **ключовата думичка function**, след това **име на функцията, списък от параметри**, ограден с кръгли скоби **()** и накрая **тяло на функцията**, оградено с фигурни скоби **{}**. Особеното в случая е, че **името на функцията** не е задължителен елемент, но е препоръчително да свикнете да

го добавяте. В примерът програмата ще работи без проблеми, дори и да пропуснем да изпишем името `getSquareFunc`. Ако пропуснем името, функцията се нарича **анонимна**.

Когато декларираме дадена променлива в тялото на една функция (чрез ключовата думичка `let` или `const`), я наричаме **локална** променлива за функцията. 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 функцията. Тази област се нарича **област на видимост** на променливата (variable scope).

## Декларативно или експресивно

Разликата между деклариране на функция чрез **декларация** или **експресия** е сравнително проста. Всички функции, декларирани чрез **function declaration**, се зареждат в паметта на програмата преди да започне нейното изпълнение, докато програмата разбира за функции, декларирани с **function expression** едва когато започне да се изпълнява и стигне до реда, на който е декларирана функцията.

На практика това означава, че можете да **извикате функция**, декларирана с **function declaration** дори и в някои от предните редове - преди нейната декларация, докато ако опитате да направите това с **function expression**, програмата ще ви **даде грешка**, че не разпознава тази функция по време на изпълнението.

## Извикване на функции

Извикването на функции представлява **стартнирането на изпълнението на кода**, който се намира в **тялото на функцията**. Това става като изпишем **името** на функцията, последвано от кръглите скоби `()` и знака `;` за край на реда. Ето един пример:

```
printHeader();
```

Дадена функция може да бъде извикана от **няколко места** в нашата програма. Единият начин е да бъде извикана от **главната област на програмата** (global scope).

```
function printHeader() {
    console.log("-----");
}

printHeader();
```

Функция може да бъде извикана и от **тялото на друга функция**, която **не** е главната област на програмата ни:

```
function printHeader() {
    console.log("-----");
}
```

```
function printMessage() {
    // invoking function from the body of another function
    printHeader();
}
```

Съществува вариант функцията да бъде извикана от **собственото си тяло**. Това се нарича **рекурсия** и можете да намерите повече информация за нея в [Wikipedia](#) или да потърсите сами в Интернет.

## Пример: празна касова бележка

Да се напише функция, която печата празна касова бележка. Функцията трябва да извиква други три функции: една за принтиране на заглавието, една за основната част на бележката и една за долната част.

Част от касовата бележка	Текст
Горна част	CASH RECEIPT -----
Средна част	Charged to _____ Received by _____
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 функция за **принтиране на заглавната част** от касовата бележка (header). Нека ѝ дадем смислено име, което описва кратко и ясно целта ѝ, например **printReceiptHeader**. В тялото ѝ ще напишем следното:

```
function printReceiptHeader() {
    console.log("CASH RECEIPT");
    console.log("-----");
}
```

Съвсем аналогично ще създадем още две функции 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 една функция**, която ще извиква трите функции, които написахме до момента една след друга:

```
function printReceipt() {
    printReceiptHeader();
    printReceiptBody();
    printReceiptFooter();
}
```

Накрая ще **извикаме** функцията **printReceipt** от global scope-а на нашата програма:

```
printReceipt();
```

## Тестване в Judge системата

Програмата с общо четири функции, които се извикват една от друга, 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/943#0>.

## Функции с параметри

Много често в практиката, за да бъде решен даден проблем, функцията, с чиято помощ постигаме това, се нуждае от **допълнителна информация**, която зависи от задачата ѝ. Именно тази информация представляват **параметрите на функцията** и нейното поведение зависи от тях.

## Използване на параметри във функции

Ако функцията ни изиска **входни данни**, то те се подават в скобите **( )**, като последователността на **фактическите параметри** трябва да съвпада с последователността на подадените при декларирането на функцията. Както отбелязахме по-горе, **параметрите освен нула на брой, могат също така да са един или няколко**. При декларацията им ги разделяме със запетая.

Декларираме функцията **printNumbers(...)** и списъка с параметри, от които тя се нуждае, за да работи коректно, след което пишем кода, който ще изпълнява:

```
function printNumbers(start, end) {
    for (let i = start; i <= end; i += 1) {
        console.log(i);
    }
}
```

След това **извикваме** функцията, като ѝ **подаваме конкретни стойности**:

```
printNumbers(5, 10);
```

При **декларирането** на параметри трябва да внимаване всеки един параметър да има **име**. Важно е при извикване на функцията, да подаваме **стойности** за параметрите по **реда**, в който са **деклариирани** самите те. В примера, който разглеждахме на променливата **start** ще бъде присвоена стойността на първият подаден параметър - в нашият случай числото 5. На променливата **end** ще бъде присвоена стойността на вторият параметър, който сме подали - в случая числото 10.

Важно е да се отбележи, че в езикът **JavaScript** декларирането на функция с даден **брой параметри**, не ни задължава да извикваме функцията със **същият брой параметри**. Можем да извикаме функция като и подадем както **повече**, така и **по-малко** параметри, като това няма да доведе до грешка.

Нека разгледаме следния пример:

```
function printNumbers(start, end) {
  for (let i = start; i <= end; i += 1) {
    console.log(i);
  }
}
```

```
printNumbers(5, 10, 15, 20);
```

В случая извикваме функцията **printNumbers(...)** като и подаваме 4, вместо **декларираните** 2 параметъра. Всички излишни параметри ще бъдат **игнорирани**. Т.е. числата 15 и 20, няма да стигнат до функцията, защото нямаме **деклариран параметър**, който да ги приеме.

Нека разгледаме още един пример:

```
function printNumbers(firstNumber, secondNumber) {
  console.log(firstNumber);
  console.log(secondNumber);
  // this will print "undefined" on the console
}

printNumbers(5);
```

В случая извикваме функцията **printNumbers(...)** като и подаваме 1, вместо **декларираните** 2 параметъра. Всички параметри, за които **не е подадена стойност**, ще получат автоматично стойност **undefined**. В нашият случай променливата **secondNumber** ще има стойност **undefined**.

## Пример: знак на цяло число

Да се създаде функция, която 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 **декларирането** на функция и даването ѝ на описателно име, например **printSign**. Тази функция ще има само един параметър:

```
function printSign(num) {  
}
```

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 функцията.

Следващата ни стъпка е да извикаме функцията, която създадохме:

```
function solve([n]) {  
    function printSign(num) {  
        // TODO  
    }  
  
    const num = parseInt(n);  
  
    printSign(num);  
}
```

## Тестване в Judge системата

Тествайте решението си: <https://judge.softuni.bg/Contests/Practice/Index/943#1>.

## Незадължителни параметри

Езикът JavaScript поддържа използването на **незадължителни** параметри. Те позволяват **пропускането** на параметри при извикването на функцията. Декларирането им става чрез **осигуряване на стойност по подразбиране** в описанието на съответния параметър.

Следващият пример онагледява употребата на незадължителните параметри:

```
function printNumbers(start = 0, end = 10) {
    for (let i = start; i <= end; i += 1) {
        console.log(i);
    }
}
```

Показаната функция `printNumbers(...)` може да бъде извикана по няколко начина:

```
printNumbers();
printNumbers(2);
printNumbers(2, 3);
```

При липсата на подадена стойност на параметър, той ще приеме стойността, с която сме го декларирали при декларацията на функцията.

## Пример: принтиране на триъгълник

Да се създаде функция, която принтира триъгълник с `n` реда, 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

Избираме смислено име за функцията, което описва целта ѝ, например `printLine`, и я имплементираме:

```
function printLine(start = 1, end) {
    let line = "";

    for (let i = start; i <= end; i += 1) {
        line += i + " ";
    }

    console.log(line);
}
```

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 (let i = 1; i < n; i++) {
    printLine(1, i);
}
```

След това разпечатваме средната линия:

```
printLine(1, num);
```

Накрая разпечатваме долната част от триъгълника, като този път стъпката на цикъла намалява.

```
for (let i = n-1; i > 0; i--) {
    printLine(1, i);
}
```

## Тестване в Judge системата

Тествайте решението си: <https://judge.softuni.bg/Contests/Practice/Index/943#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

Създаваме функция, която 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 вградения метод **repeat(...)**:

```
function printHeaderFooter(num) {
    console.log("-".repeat(2 * num));
}
```

Следващата ни стъпка е да създадем функция, която ще рисува на конзолата средните редове. Отново задаваме описателно име, например **printMiddleRow**:

```
function printMiddleRow(num) {
    let line = "-";

    for (let i = 0; i < num - 1; i++) {
        line += "\\/";
    }

    line += "-";

    console.log(line);
}
```

Накрая извикваме създадените функции за да нарисуваме целия квадрат:

```
function solve([n]) {
    const num = parseInt(n);

    printHeaderFooter(num);
    // TODO: Draw the rest of the square
}
```

## Тестване в Judge системата

Тествайте решението си: <https://judge.softuni.bg/Contests/Practice/Index/943#3>.

## Връщане на резултат от функция

До момента разгледахме функции, които извършват дадено действие, например отпечатване на даден текст, число или фигура на конзолата. Освен този тип функции, съществуват и такива, които могат да **връщат** някакъв **резултат** от своето изпълнение - например резултатът от умножението на две числа. Именно тези функции ще разгледаме в следващите редове.

## Оператор return

За да получим резултат от функцията, на помощ идва операторът **return**. Той трябва да бъде **използван в тялото** на функцията и указва на програмата да **спре изпълнението** на функцията и да **върне** на извиквача определена **стойност**. Тази стойност се определя от израза след въпросния оператор **return**.

В примера по-долу имаме **функция**, която получава име и фамилия като **параметри**, съединява ги и **връща** като резултат пълното име.

```
function getFullName(firstName, lastName) {
    return firstName + " " + lastName;
}
```

Има случаи, в които **return** може да бъде извикван от няколко места във функцията, но само ако има **определенi** входни условия.

В примера по-долу имаме функция, която 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. Функцията използва ключовата дума **return** на три различни места, за да върне три различни стойности според логиката на сравненията на числата:

```
function compareTo(number1, number2) {
    if (number1 > number2) {
        return 1;
    } else if (number1 === number2) {
        return 0;
    } else {
        return -1;
    }
}
```

### Кодът след **return** е недостъпен

В случай, че **return** операторът не се намира в условна конструкция като **if**, след него, в текущия блок, **не** трябва да има други редове код, тъй като тогава Visual Studio Code ще покаже предупреждение, съобщавайки ни, че е засякъл код, който не може да бъде достъпен:

```
function getFullName(firstName, secondName, familyName) {
    return firstName + " " + secondName;

    [jshint] Unreachable 'return' after 'return'. (W027)
    return firstName + " " + secondName + " " + familyName;
}
```

Операторът **return** може да бъде използван и без да бъде специфицирана **конкретна стойност**, която да бъде върната. В този случай, просто ще бъде прекратено изпълнението на кода във функцията и ще бъде върната стойност по подразбиране **undefined**.



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 върнатата от функцията стойност

След като дадена функция е изпълнена и върне стойност, то тази стойност може да се използва по **няколко** начина.

Първият е да присвоим резултата като стойност на променлива:

```
let max = getMax(5, 10);
```

Вторият е резултатът да бъде използван в израз:

```
let total = getPrice() * quantity * 1.20;
```

Третият е да подадем резултата от работата на функцията към **друга функция**:

```
let age = parseInt(getMyAge());
```

### Пример: пресмятане на лицето на триъгълник

Да се напише функция, която 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

Създаваме функция, с коректно име.

```
function getTriangleArea(a, b) {
    return (a * b) / 2;
}
```

Следващата ни стъпка е да извикаме новата функция и да запишем върнатата стойност в подходяща променлива:

```
function solve([length, height]) {
    const a = parseFloat(length);
    const b = parseFloat(height);
    const area = getTriangleArea(a, b);
    console.log(area);
}
```

## Тестване в Judge системата

Тествайте решението си: <https://judge.softuni.bg/Contests/Practice/Index/943#4>.

### Пример: степен на число

Да се напише функция, която изчислява и връща резултата от повдигането на число на дадена степен.

#### Примерен вход и изход

Вход	Изход	Вход	Изход
2 8	256	3 4	81

#### Насоки и подсказки

Първата ни стъпка отново ще е да създадем функция, която ще приема два параметъра (числото и степента) и ще връща като резултат числото повдигнато на съответната степен.

```
function calculatePower(num, power) {
    let result = 0;
    // TODO: Calculate result
    // Use a loop or Math.pow()
    return result;
}
```

След като сме направили нужните изчисления, ни остава да извикаме декларираната функция.

## Тестване в Judge системата

Тествайте решението си: <https://judge.softuni.bg/Contests/Practice/Index/943#5>.

### Функции, връщащи няколко стойности

В практиката се срещат случаи, в които се нуждаем дадена функция да върне повече от един елемент като резултат. В езикът **JavaScript** има 2 начина по които може да бъде постигнато това - чрез **деструкция** и чрез **връщане на обект**.

#### Деструкция

Когато желаем функция да върне **повече от една стойност**, използваме ключовата думичка **return**, след което изброяваме всички стойности, които желаем да върнем, като ги ограждаме в квадратни скоби - **[ , ]**:

```
function getNames(firstName, secondName, familyName) {
    const name = firstName + " " + familyName;
    const fullName =
        firstName + " " + secondName + " " + familyName;
    return [name, fullName];
}
```

След това за да получим върнатите стойности, отново на помощ идват квадратните скоби. Изброяваме параметри, които да получат тези стойности, като присвояването ще стане по реда, по който стойностите са върнати:

```
let [name, fullName] = getNames("John", "Silver", "Doe");
```

В горният пример променливата **name** ще получи стойността "John Doe", която е първата върната стойност от функцията **getNames**, а **fullName** ще получи "John Silver Doe", която е втората върната стойност.

## Обекти

Този подход е много подобен на предния, като разликата е, че не просто изброяваме стойностите, които искаме да върнем, но и им даваме имена. Обектите са изключително важна и основна част от езика **JavaScript**. За сега е достатъчно да знаете, че се декларират с фигурните скоби **{ }** , като между тях изброяваме името на стойността (нарича се **ключ**), последвано от знака две точки **:** и самата стойност. Разделяме отделните двойки **ключ-стойност** със символа **,**:

```
return {
    name: firstName + " " + familyName,
    fullName: firstName + " " + secondName + " " + familyName
};
```

В този пример връщаме обект, който държи 2 стойности - **name** и **fullName**:

```
function getNames(firstName, secondName, familyName) {
    return {
        name: firstName + " " + familyName,
        fullName: firstName + " " + secondName + " " + familyName
    };
}

const personNames = getNames("John", "Doe", "Silver");
```

Тук променливата **personNames** ще получи всички върнати стойности. Като **name** и **fullName** са част от тези стойности и могат да бъдат достъпени със символа **.**:

```
console.log(personNames.name);
console.log(personNames.fullName);
```

## Варианти на функции

В много езици за програмиране една и съща функция може да е декларирана в **няколко варианта** с еднакво име и различни параметри. Това е известно с термина "method overloading". За добро или лошо **JavaScript** не поддържа тази възможност.

Когато декларирате **две или повече функции с еднакви имена**, програмата ще използва **последно декларираната** такава. Декларирайки втора функция със същото име, вие реално премахвате старата функция и записвате на нейно място новата.

## Вложени функции

Нека разгледаме следния пример:

```
function solve() {
    function sum(a, b) {
        return a + b;
    }

    const a = 10;
    const b = 20;

    console.log(sum(a, b));
}
```

### Какво е локална функция?

Виждаме, че в този код, в нашата функция **solve()** има друга декларирана функция **sum()**. Такава **вложена** функция се нарича **локална** функция. Локалните функции могат да се декларират във всяка една друга функция.

### Зашто да използваме локални функции?

С времето и практиката ще открием, че когато пишем код, често се нуждаем от функции, които бихме използвали само един път, или пък нужната ни функция става твърде дълга. По-нагоре споменахме, че когато една функция 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 дадена функция да се декларира друга функция, която ще бъде използвана 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.

### Деклариране на локални функции

Нека отново разгледаме примера от по-горе.

```
function solve() {
    function sum(a, b) {
        return a + b;
    }

    const a = 10;
    const b = 20;

    console.log(sum(a, b));
}
```

В този пример, функцията `sum()` е локална функция, тъй като е вложена във функцията `solve()`, т.е. `sum()` е локална за `solve()`. Това означава, че функцията `sum()` може да бъде използван само във функцията `solve()`, тъй като е декларирана в нея.

Локалните функции имат достъп до променливи, които са деклариирани на същото или по-горно ниво от тях. Следващият пример демонстрира това:

```
function solve() {
    const message = "I will be used in local function";

    function printMessage() {
        console.log(message);
    }

    printMessage();
}
```

Тази особеност на **вложените функции** ги прави много удобни помощници при решаването на дадена задача. Те спестяват време и код, които иначе бихме вложили, за да предаваме на вложените функции параметри и променливи, които се използват във функциите, в които са вложени.

## Именуване на функции. Добри практики при работа с функции

В тази част ще се запознаем с някои **утвърдени практики** при работа с функции, свързани с именуването, подредбата на кода и неговата структура.

### Именуване на функции

Когато именуваме дадена функция е препоръчително да използваме **смислени имена**. Тъй като всяка функция **отговаря** за някаква част от нашия проблем, то при

именуването ѝ трябва да вземем предвид действието, което тя извършва, т.е. добра практика е **името да описва целта**.

Задължително е името да започва с **малка буква** и трябва да е съставено от глагол или от двойка: глагол + съществително име. Форматирането на името става, спазвайки **Lower Camel Case** конвенцията, т.е. **всяка дума, с изключение на първата, започва с главна буква**. Кръглите скоби ( и ) винаги следват името на функцията.

Всяка функция трябва да изпълнява самостоятелна задача, а името на функцията трябва да описва каква е нейната роля.

Няколко примера за **коректно** именуване на функции:

- **findStudent**
- **loadReport**
- **sine**

Няколко примера за **лошо** именуване на функции:

- **method1**
- **doSomething**
- **handleStuff**
- **sampleMethod**
- **dirtyHack**
- **FindStudent**
- **LoadReport**

Ако не можем да измислим подходящо име, то най-вероятно функцията решава повече от една задача или няма ясно дефинирана цел и тогава трябва да помислим как да я разделим на няколко отделни функции.

## Именуване на параметрите на функциите

При именуването на **параметрите** на функцията важат почти същите правила, както и при самите функции. Разликите тук са, че за имената на параметрите е добре да използваме съществително име или двойка от прилагателно и съществително име. 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 функции:

- **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 функции

Нека отново припомним, че една функция трябва да изпълнява **само една** точно определена **задача**. Ако това не може да бъде постигнато, тогава трябва да помислим как да **разделим** функцията на няколко отделни такива. Както казахме, името на функцията трябва точно и ясно да описва нейната цел. Друга добра практика в програмирането е да **избягваме** функции, по-дълги от екрана ни (приблизително). Ако все пак кода стане много обемен, то е препоръчително функцията да се **раздели** на няколко по-кратки, както в примера по-долу.

```
function printReceipt() {
    printHeader();
    printBody();
    printFooter();
}
```

## Структура и форматиране на кода

При писането на функции трябва да внимаваме да спазваме коректна **индентация** (отместване по-навътре на блокове от кода).

Пример за **правилно** форматиран JavaScript код:

```
function solve() {
    // some code here...
    // some more code...
}
```

Пример за **некоректно** форматиран JavaScript код:

```
function solve() {
    // some code here...
// some more code...
}
```

Когато заглавният ред на функцията е **твърде дълъг**, се препоръчва той да се раздели на няколко реда, като всеки ред след първия се отмества с две табулации надясно (за по-добра четимост):

```
function solve(firstName, secondName,
  familyName) {
  // some code here...
  // some more code...
}
```

Друга добра практика при писане на код е да **оставяме празен ред** между функциите, 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 функции:

- Научихме, че **целта** на функциите е да **разделят** големи програми с много редове код на по-малки и кратки задачи.
- Запознахме се със **структурата** на функциите, как да ги **декларираме** и **извикваме** по тяхното име.
- Разгледахме примери за функции с **параметри** и как да ги използваме в нашата програма.
- Научихме какво представляват **сигнатурата** и **връщаната стойност** на функцията, както и каква е ролята на оператора **return**.
- Запознахме се с **добрите практики** при работа с функции, как да именуваме функциите и техните параметри, как да форматираме кода и други.

## Упражнения

За да затвърдим работата с функции, ще решим няколко задачи. В тях се изисква да напишете функция с определена функционалност и след това да я извикате като ѝ подадете данни, точно както е показано в примерния вход и изход.

### Задача: "Hello, Име!"

Да се напише функция, която получава като параметър име и принтира на конзолата "Hello, !!".

### Примерен вход и изход

Вход	Изход
Peter	Hello, Peter!

## Насоки и подсказки

Дефинирайте функция **printName(name)** и я имплементирайте. Да се напише функция **solve(...)**, която получава като входни данни име на човек и извиква **printName** функцията като подава прочетеното име.

## Тестване в Judge системата

Тествайте решението си: <https://judge.softuni.bg/Contests/Practice/Index/943#7>.

## Задача: по-малко число

Да се създаде функция **getMin(a, b)**, която връща по-малкото от две числа. Да се напише функция **solve(...)**, която получава като входни данни три числа и печата най-малкото от тях. Да се използва функцията **getMin(...)**, която е вече създадена.

## Примерен вход и изход

Вход	Изход	Вход	Изход
1		-100	
2	1	-101	
3		-102	-102

## Насоки и подсказки

Дефинирайте функция **getMin(int a, int b)** и я имплементирайте, след което я извикайте от функцията **solve(...)**, 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:

```
function solve([num1, num2, num3]) {
    let min = getMin(getMin(num1, num2), num3);
}
```

## Тестване в Judge системата

Тествайте решението си: <https://judge.softuni.bg/Contests/Practice/Index/943#8>.

## Задача: повтаряне на низ

Да се напише функция **repeatString(str, count)**, която получава като параметри стрингова променлива **str** 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 функцията по-долу като добавите входния низ към резултата в цикъла:

```
function repeatString(str, count) {
    let repeatedString = "";

    for (let i = 0; i < count; i += 1) {
        // TODO
    }

    return repeatedString;
}
```

## Тестване в Judge системата

Тествайте решението си: <https://judge.softuni.bg/Contests/Practice/Index/943#9>.

## Задача: n-та цифра

Да се напише функция **findNthDigit(number, index)**, която 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 отпечатва на конзолата.

## Примерен вход и изход

Вход	Изход	Вход	Изход	Вход	Изход
83746 2	4	93847837 6	8	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/943#10>.

### Задача: число към бройна система

Да се напише функция **integerToBase(number, toBase)**, която 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	Вход	Изход	Вход	Изход
3 2	11	4 4	10	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 във функцията по-долу:

```
function integerToBase(number, 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/943#11>.

### Задача: известия

Да се напише функция **solve(...)**, която приема като първи параметър цяло число **n** - брой на съобщения и допълнителен брой параметри, които са самите части

на **съобщенията**. За всяко съобщение може да се получи различен брой параметри. Първият параметър за всяко съобщение е **messageType**, който може да бъде **success**, **warning** или **error**:

- Когато **messageType** е **success**, следващите 2 получени параметъра са **operation** и **message**
- Когато **messageType** е **warning** следващият параметър е **message**.
- Когато **messageType** е **error** следващите 3 получени параметъра са **operation + message + errorCode**(всяко е отделен параметър).

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 дефиниране на четири функции: **showSuccessMessage(...)**, **showWarningMessage(...)**, **showErrorMessage(...)** и **processMessage(...)**, като само последната функция да се извика от главната **solve(...)** функция:

```
function showSuccessMessage(operation, message) {}
function showWarningMessage(message) {}
function showErrorMessage(operation, message, errorCode) {}
function processMessage(messageInfo) {}
```

### Примерен вход и изход

Вход	Изход
4 <b>error</b> credit card purchase Invalid customer address 500	Error: Failed to execute credit card purchase. =====Reason: Invalid customer address. Error code: 500.
<b>warning</b> Email not confirmed	Warning: Email not confirmed. =====
<b>success</b> user registration User registered successfully	Successfully executed user registration. =====User registered successfully.
<b>warning</b> Customer has not email assigned	Warning: Customer has not email assigned. =====

### Насоки и подсказки

Дефинирайте и имплементирайте четирите функции. След това извикайте функцията **processMessage(...)** от главната **solve(...)** функция:

```
function solve(messageInfo) {
    processMessage(messageInfo);
}
```

В **processMessage(...)** извадете първо броят на съобщенията и след това ги обработете едно по едно спрямо техният тип и извикайте съответната функция за печтане.

## Тестване в Judge системата

Тествайте решението си: <https://judge.softuni.bg/Contests/Practice/Index/943#12>.

## Задача: числа към думи

Да се напише функция **letterize(number)**, която получава 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	Вход	Изход
3 999 -420 1020	nine-hundred and ninety nine minus four-hundred and twenty too large	2 15 350	fifteen three-hundred and fifty

Вход	Изход	Вход	Изход
4 311 418 509 -9945	three-hundred and eleven four-hundred and eighteen five-hundred and nine too small	3 500 123 9	five-hundred one-hundred and twenty three 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 отпечатаваме "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 отпечатаваме "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 отпечатаваме "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/943#13>.

## Задача: криптиране на низ

Да се напише функция **encrypt(char letter)**, която 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.
- Функцията трябва да върне като резултат криптириания низ.

Пример:

- **j → 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 дефинира функция **solve(...)**, която получава **поредица от символи, криптира ги** и отпечатва резултата на един ред. Приемаме, че входните данни винаги ще бъдат валидни. **solve(...)** функцията трябва да получава входните данни, подадени от потребителя – цяло число **n**, последвано от по един символ за всеки следващ **n** елемент. Да се криптират символите и да се добавят към криптирания низ. Накрая като резултат трябва да се отпечата **криптиран низ от символи** като в следващия пример.

**Пример:**

- S, o, f, t, U, n, i → V83Kp11nh12ez16sZ85Mn10mn15h

### Примерен вход и изход

Вход	Изход
4 s   a p	x15rt18kh97Xr12o

Вход	Изход
7 S o f t U n i	V83Kp11nh12ez16sZ85Mn10mn15h

### Насоки и подсказки

На променливата **result**, в която ще се пази стойността на резултата, ще присвоим първоначална стойност **""**. 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".

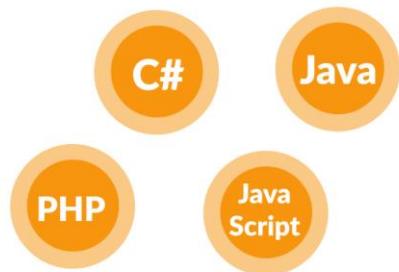
### Тестване в Judge системата

Тествайте решението си: <https://judge.softuni.bg/Contests/Practice/Index/943#14>.

## 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 **профессионалисти 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 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**, съдействайки 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](http://softuni.bg/apply)

# Глава 11. Хитрости и хакове

В настоящата глава ще разгледаме някои хитрости, хакове и техники, които ще улеснят работата ни с езика **JavaScript** в среда за разработка **Visual Studio Code**. 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 разбирам**, 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 да 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 разбирам.

Езикът **JavaScript** е създаден от **Брендън Айх** като част от развитието на един от първите браузъри **Netscape**. Основните конструкции и базов синтаксис преднамерено **приличат на Java**, за да се намалят усилията за научването им. Още повече, дори се използват подобни конвенции за писане и форматиране на кода. Трябва да знаете, че дори да не спазвате наложените конвенции, кодът ще **работи** (стига да е написан правилно), но просто **няма да бъде лесно разбирам**. Това, разбира се, не е фатално на основно ниво, но колкото по-бързо свикнете да пишете качествен код, толкова по-добре.

Правилата, които се ползват при писане на **JavaScript** могат да бъдат намерени на много места. Официалните правила или т.н. **JavaScript код конвенции** са описана много добре в статията "**Coding style**" в документацията на "Mozilla": [https://developer.mozilla.org/en-US/docs/Mozilla/Developer\\_guide/Coding\\_Style](https://developer.mozilla.org/en-US/docs/Mozilla/Developer_guide/Coding_Style).

Важно е да се отбележи, че в примерите, които сме давали до сега и ще даваме занапред в тази книга, се ръководим основно от нея.

За форматиране на кода се препоръчва **къдрявите скоби {}** да се отварят на същия ред и да се затварят точно под конструкцията, към която се отнасят, както е в примера по-долу.

```
if (someCondition) {
```

```

        console.log("Inside the if statement");
    }

```

Вижда се, че командалата `console.log(...)` в примера е **4 празни полета навътре** (**един таб**), което също се препоръчва в документацията. Също така, 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.log("Inside the if statement");
    }
}

```

Ето това е пример за **лошо форматиран код** спрямо общоприетите конвенции за писане на код на езика `JavaScript`:

```

if(someCondition)
{
    console.log("Inside the if statement");}

```

Първото, което се забелязва са **къдравите скоби {}**. Първата (отваряща) скоба трябва да е **точно до if** условието, а втората (затваряща) скоба - **под командалата console**

**.log(...)**, на отделен празен ред. В допълнение, командалата 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 циклите**, както и всякакви **други конструкции** с къдрави скоби {}. Ето още няколко примера:

Правилно:

```

for (let i = 0; i < 5; i++) {
    console.log(i);
}

```

Грешно:

```

for(let i=0;i<5;i++)
{
    console.log(i);
}

```

За ваше удобство има **бързи клавиши** във **Visual Studio Code**, за които ще обясним по-късно в настоящата глава, но засега ни интересуват следните комбинации за форматиране на **кода в целия документ**:

- За Windows [Shift + Alt + F]
- За Mac [Shift + Option + F]
- За Ubuntu [Ctrl + Shift + I]

Нека използваме **грешния пример** от преди малко:

```
for(let i=0;i<5;i++)
{
  console.log(i);
}
```

Ако натиснем [Shift + Alt + F], което е нашата комбинация за форматиране на **целия документ**, ще получим код, форматиран според **общоприетите конвенции за JavaScript**. Автоматичното форматиране, обаче не влияе на именуването на нашите променливи, за което ние трябва да се погрижим сами.

## Именуване на променливи

Основното правило за имената на променливите в програмирането е, че **“името на една променлива трябва да обяснява накратко нейното предназначение”**. Например, променлива, с която броим цифри в текст, може да има име **lettersCount**.

По подобен принцип се именуват и **функциите** в програмирането: да отговарят на въпроса **“какво действие извършва тази функция”**. Съответно най-често в името има глагол + съществително, например **drawLine(size)**.

Прието е променливите в JavaScript 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 **JavaScript прави разлика** между тях. Например **age** и **Age** са различни променливи.
- Имената на променливите **не могат да съвпадат със служебна дума** (keyword) от езика JavaScript, например **let** е невалидно име на променлива. Служебните или т.н. ключови фрази са просто думи, които са **част от синтаксиса на JavaScript** и поради тази причина те са резервирали и не могат да бъдат ползвани като имена на нашите променливи. Чрез тези думи ние имаме възможност да изграждаме нашите програми. Като пример за такива думи могат да се дадат вече използваните: **for, while, do, if, else, let** и др. Пълен списък с тези резервирали фрази можете да видите тук: [https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Lexical\\_grammar#Keywords](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Lexical_grammar#Keywords)



Въпреки че използването на символа `_` в имената на променливите е разрешено, в **JavaScript** това **не се препоръчва** и се счита за лош стил на именуване.

Ето няколко примера за **добре именувани** променливи:

- `firstName`
- `age`
- `startIndex`
- `lastNegativeNumberIndex`

Ето няколко примера за **лошо именувани променливи**, макар и имената да са коректни от гледна точка на езика JavaScript:

- `_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 Code

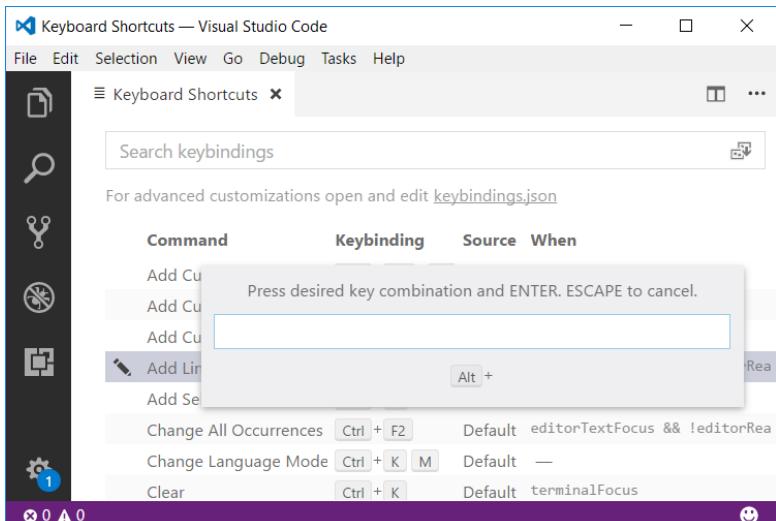
В предната секция споменахме за някои от комбинациите, които се отнасят за форматиране на код. Едната комбинация [`Shift + Alt + F`] беше за **форматиране на целия код в даден файл**, а другите правиха същото нещо но на различна операционна система. Тези комбинации се наричат **бързи клавиши** и сега 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 Code**. Изброените клавишни комбинации работят със сигурност и са изпробвани на Windows. Идеята е да ви покажем, че това съществува, ползва се лесно и при нужда винаги можете да намерите, това което ви трябва за всяка една операционна система.

Комбинация	Действие
[CTRL + F]	Комбинацията отваря търсачка, с която можем да търсим в нашия код.
[CTRL + /]	Закоментира част от кода и съответно премахва коментара от закоментиран код
[CTRL + Z]	Връща една промяна назад (т.нар. Undo).
[CTRL + Y]	Комбинацията има противоположно действие на [CTRL + Z] (т.нар. Redo).
[Shift + Alt + F]	Форматира кода според конвенциите по подразбиране.
[CTRL + Backspace]	Изтрива думата вляво от курсора.
[CTRL + Del]	Изтрива думата вдясно от курсора.
[CTRL + K S]	Запазва всички файлове в проекта.
[CTRL + S]	Запазва текущия файл.

Повече за бързите клавиши във Visual Studio Code може да намерите тук: <https://code.visualstudio.com/shortcuts/keyboard-shortcuts-windows.pdf>.

Ако пък вече се чувствате достатъчно уверени в уменията си с бързите клавиши, отворете Visual Studio Code, натиснете [CTRL + K + S] (обърнете внимание, че е различно от [CTRL + K S], при което Ctrl и K се натискат едновременно, а S след това), при което ще се отвори прозорец в самата среда за разработка, който съдържа пълен списък с всички възможни клавишни комбинации в света на Visual Studio Code. Още повече, дори от там ще можете да правите промени по съществуващите клавишни комбинации:



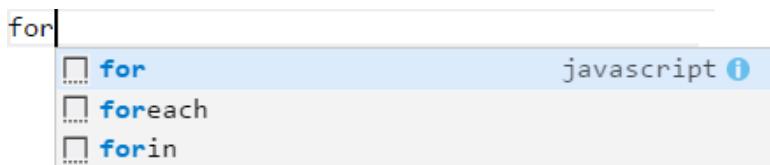
Не се колебайте, приложите наученото сега и използвайте клавишните комбинации, които мислите, че ще ви спомогнат в писането на вашите програми!

## Шаблони с код (code snippets)

Във Visual Studio Code съществуват т.нар. **шаблони с код** (code snippets), при изписването на които се изписва по шаблон някакъв блок с код. Тази полезна опция не е включена по подразбиране. Вие сами трябва да я активирате от [File -> Preferences -> Settings] (или просто [Ctrl + Comma]), при което ви се отваря прозорец наречен **User Settings**. Това са вашите лични настройки, които много лесно можете да промените. Просто добавете следния ред между отварящата и затварящата къдрави скоби в дясната част на екрана:

```
"editor.tabCompletion": true
```

След като направите това, при изписването на "**for**" и натискане на [Tab] + [Tab], автоматично се генерира кодът на **цилостен for цикъл** в тялото на нашата програма. Това се нарича "разгъване на шаблон за кратък код". Подобно работи и шаблона "**if**" + [Tab] + [Tab]. На фигурата по-долу е показано действието на шаблона "**for**":



## 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 **json** обект. Като за начало ще отидем на [File -> Preferences -> User Snippets], след което ще се отвори прозорец, от който да си изберете за кой език за програмиране ще създавате шаблон, както е показано на картинката.

**Избираме JavaScript** от падащото меню и ще се отвори прозорец с наименуван **javascript.json**. Това разширение **json** е един специален формат на записване на данни, който е наложен при начините за пренос и запазване на данни. Освен това, **json** форматът може да се ползва и в нашите програми, както ще разгледаме малко по - късно. Файлът изглежда така:



```
{
/*
// Place your snippets for JavaScript here. Each snippet is defined
under a snippet name and has a prefix, body and
// description. The prefix is what is used to trigger the snippet
and the body will be expanded and inserted. Possible variables are:
// $1, $2 for tab stops, $0 for the final cursor position, and $
{1:label}, ${2:another} for placeholders. Placeholders with the
// same ids are connected.
// Example:
"Print to console": {
    "prefix": "log",
    "body": [
        "console.log('$1');",
        "$2"
    ],
    "description": "Log output to console"
}
*/
}
```

Примерът, който виждаме по подразбиране генерира код за писане по конзолата чрез ключовия префикс `log`. Този код е само примерен и всъщност този шаблон е вграден, но ако не беше, би изглеждал като примера.

В този пример виждате доста непознати неща, но няма страшно, по-нататък ще се запознаем и с тях. Сега се фокусираме върху частта **"Print to console"**: и кода между **отварящата и затварящата къдрани скоби {}**. Това, което виждаме вътре в скобите представлява съдържанието на един шаблон. Всеки шаблон трябва да съдържа **prefix**, който представлява краткия текст, който след като натиснете **[Tab]** + **[Tab]** ще създава кода на шаблона във вашата програма.

Второто нещо, което трябва да има вашият шаблон е **body**, това е най-сложната част от шаблона. Това всъщност е **кодът, който ще се генерира**, като в него може да използваме **променливи**, които се създават с **\$1**, като на мястото на единицата може да бъде поставен друг текст. В примера е използвана променлива: **"console.log('\$1');"**.

Може да използваме **Tab stops**, които просто поставят курсора на определени места в кода и между тях може да се навигира с табулация. Те се създават автоматично чрез създаване на променлива. Може да използваме и **Placeholders**, те представляват вид **Tab stops**, но могат да съдържат и някаква стойност, например: **`\${1:myVal}`**.

Съществуват и по сложни конфигурации, но като за начало тези ще ни свършат отлична работа.

Последната част от шаблона е **description**, което служи за добавяне на допълнително пояснение за това, което прави той. Сега нека да пробваме да

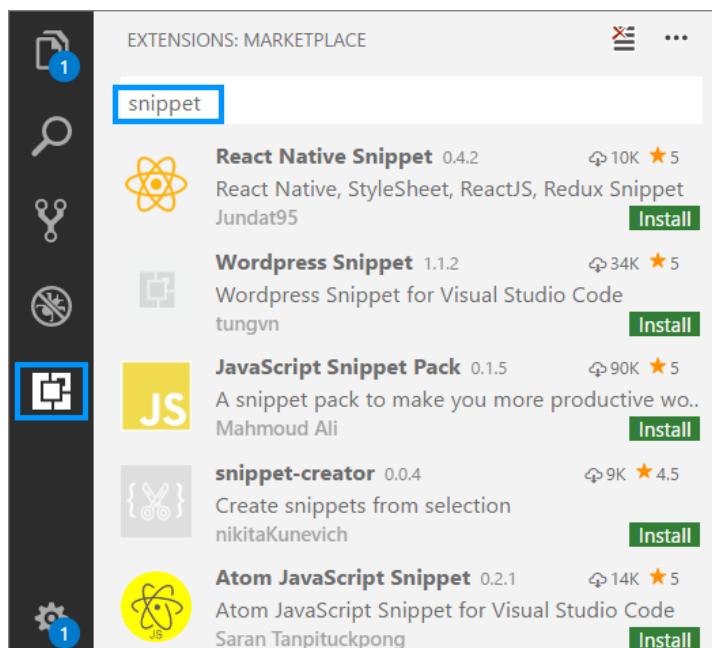
направим собствен шаблон. Изтриваме дадения пример и въвеждаме следния код:

```
{
  "Generates JSON Object": {
    "prefix": "json",
    "body": [
      "let myJson = {",
      "\t${1:myKey}: '${2:Value}'",
      "}";
    ],
    "description": "Generates JSON Object in our program"
  }
}
```

Вече когато напишем **json** + [Tab] + [Tab] в отворен JavaScript файл във Visual Studio Code, **нашият нов snippet** се появява:

```
let myJson = {
  myKey: 'Value'
};
```

За тези от вас, които се интересуват повече от темата, доста от големите frameworks като **Angular**, **React** и др. имат собствени шаблони, които могат да се инсталират от прозореца за **разширения** (Extensions). Както самото име подсказва, **framework** представлява концептуална структура, която ни помага, като дава някои неща наготово, а също така и ни предпазва да не правим големи грешки, като налага някои ограничения. Основната идея е да дава завършено решение в дадена област, което да има възможност за надграждане на всички компоненти в тази област. Част от нещата, които можете да получите наготово са и именно тези шаблони.



## 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 Code** ни предоставя **вграден дебъгер** (debugger), чрез който можем да поставяме **точки на прекъсване** (или т.нр. 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.

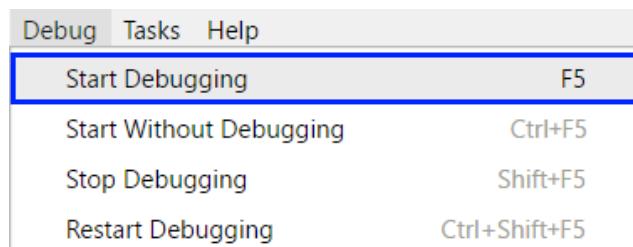
За да демонстрираме работа с дебъгера, ще използваме следната програма:

```
for (let i = 0; i < 100; i++) {
    console.log(i);
}
```

Ще сложим **стопер** (breakpoint) на метода **console.log(...)**. За целта трябва да преместим курсора на реда, който печата на конзолата, и да натиснем [F9]. Появява се **стопер** (червената точка, точно преди цифрата на ред 3), където програмата ще **спре** изпълнението си:

```
1  for (let i = 0; i < 100; i++) {
2      console.log(i)
3  }
```

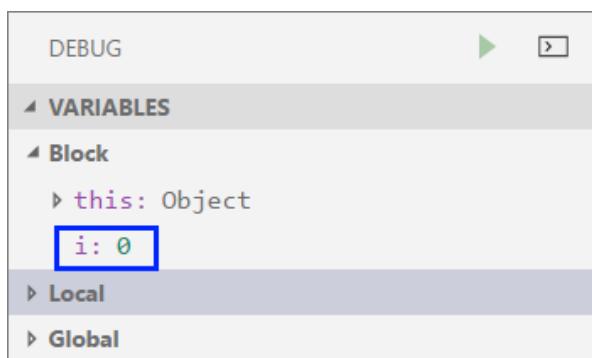
За да стартираме програмата в режим на дебъгване, избираме [Debug] -> [Start Debugging] или натискаме [F5]:



След стартиране на програмата виждаме, че тя **спира изпълнението си** на ред 4, 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 клавиш [F10]. Забелязваме, че кодът на текущия ред все още не е изпълнен. Изпълнява се, когато преминем на следващия ред:

```
1  for (let i = 0; i < 100; i++) {
2      console.log(i)
3  }
```

От прозореца **Debug**, който се отваря от [View -> Debug] или чрез клавишка комбинация [Ctrl + Shift + D], можем да наблюдаваме **промените по локалните променливи**:



## Справочник с хитrosti

В тази секция ще покажем накратко **хитrosti и техники** от програмирането с езика **JavaScript**, част от които споменавани вече в тази книга, които ще са много полезни, ако ходите на изпит по програмиране за начинаещи.

## Закръгляне на числа

При нужда от закръгляне можем да използваме един от следните методи:

- **Math.round(...)** - приема 1 параметър - **числото, което искаме да закръглим**.  
Закръглянето се извършва по основното правило за закръгляване - ако десетичната част е по-малка 5, закръгленето е надолу и обратно, ако е по-голяма от 5 - нагоре:

```
let number = 5.439;
console.log(Math.round(number));
// Това ще отпечата на конзолата "5"
let secondNumber = 5.539;
console.log(Math.round(secondNumber));
// Това ще отпечата на конзолата "6"
```

- **Math.floor(...)** - в случай, че искаме закръгленето да е винаги **надолу до предишното цяло число**. Например, ако имаме числото 5.99 и използваме **Math.floor(5.99)**, ще получим числото 5:

```
let numberToFloor = 5.99;
console.log(Math.floor(numberToFloor));
// Това ще отпечата на конзолата 5
```

- **Math.ceil(...)** - в случай, че искаме закръгленето да е винаги **нагоре до следващото цяло число**. Например, ако имаме числото 5.13 и използваме **Math.ceil(5.13)**, ще получим числото 6:

```
let numberToCeil = 5.13;
console.log(Math.floor(numberToCeil));
// Това ще отпечата на конзолата 6
```

- **Math.trunc(...)** - в случай, че искаме да **премахнем дробната част**. Например, ако имаме числото 2.63 и използваме **Math.trunc(2.63)**, ще получим 2:

```
let numberToTrunc = 2.63;
console.log(Math.floor(numberToTrunc));

// Това ще отпечата на конзолата 2
```

## В JS използвайте === вместо ==, както и !== вместо !=

В езика JavaScript операторите **==** и **!=** правят **автоматично преобразование** на сравняваната стойност или променлива, докато операторите **====** и **!==** не прави такова преобразование и ако двете стойности не са от един и същ тип - резултатът е **false**. Те (**==** и **!=**) правят сравнение на **стойността и типа**, което е по-точно и дори по-бързо. Нека да видим следния пример, чрез който да изясним какво се има в предвид под **тип** на данните:

```
[10] === 10      // false
[10] == 10       // true
"10" == 10       // true
"10" === 10      // false
[] == 0          // true
[] === 0         // false
"" == false     // true but true == "a" is false
"" === false    // false
```

Виждаме как числото **10** може да бъде записано в нашите програми по различни начини. Записано по този начин **[10]** представлява **масив** от едно число. Накратко масивите са **множество стойности** записани в дадена променлива. Например:

```
let array = [10, 20, 30, 40];
// Това е променлива от тип масив
```

В последствие ще научим повече относно масивите, но за сега нека само се замислим дали масивът **[10]** е нормално да е равен на числото **10**. Ще ви подскажем - **не е нормално**. Затова, ако не искаме неприятни грешки (бъгове) във нашите програми, най-добре да ползваме операторите **==** и **!=**.

Ситуацията с останалите оператори за сравнения е подобна и същата логика важи и там.

## 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 при изписване, може да използваме шаблонът (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 (**let i**) и се задава нейна начална стойност.
- Условие за повторение (**i <= 10**).
- Обновяване на брояча (**i++**).
- Тяло на цикъла.

```
for (let i = 0; i <= 10; i++) {
    // тяло
}
```

За улеснение при изписване, може да използваме шаблонът (code snippet) за **for** цикъл: **for** + [Tab] + [Tab]\*\*

## Използване на т.нар. положителни (**Truthy**) и отрицателни (**Falsy**) стойности

Всичките **Truthy** стойности използвани в условната конструкция **if** ще дадат положителен резултат и съответно нашата програма ще продължи изпълнението си в тялото на условната конструкция (за целите на примера тук тялото на условната конструкция не е форматирано правилно).

За част от тях изглежда логично да дават положителен резултат, но за други не чак толкова.

```
//Truthy
if (true) {}           //true
if ({}) {}             //true
if ([] {})             //true
if (42) {}             //true
if ("foo") {}          //true
if (new Date()) {}     //true
if (-42) {}            //true
if (3.14) {}           //true
if (-3.14) {}          //true
if (Infinity) {}       //true
if (-Infinity) {}      //true
```

Обратно, всички **Falsy** стойности, ще дадат отрицателен резултат и програмата няма да влезе в тялото на условната конструкция.

```
//Falsy
if (false) {}          //false
if (null) {}            //false
if (undefined) {}       //false
if (0) {}               //false
if (NaN) {}              //false
if ('') {}                //false
if ("") {}                //false
```

Не е необходимо тези стойности да се знаят наизуст на този етап, а само да се помни, че съществуват т.нр. **Truthy** и **Falsy** стойности. С времето ще свикнем как правилно да ги използваме и как да ни помагат, за да съкратим нашия код.

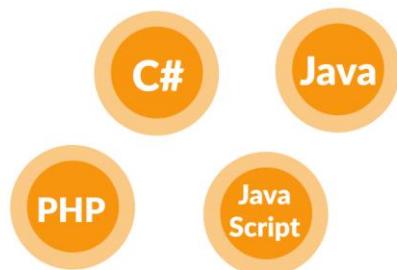
## Какво научихме от тази глава?

В настоящата глава се запознахме как **правилно да форматираме и именуваме** елементите на нашия код, някои **бързи клавиши** (shortcuts) за работа във Visual Studio Code, **шаблони с код** (code snippets) и разгледахме как се **дебъгва код**.

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](http://softuni.bg/apply)

# 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 отаряват 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 **JavaScript** 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 **JavaScript** и **Node.js** екосистемата (**Node.js**, **npm**, **Express.js** и други), **front-end** уеб технологиите (**HTML**, **CSS**, **Angular**, **React**, **AJAX**, **HTML5**) 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](#), 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 учене 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 учене 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. **Започването на работа** по специалността при силен успех в СофтУни, съчетан с желание за работа и разумни очаквания спрямо работодателя, е почти гарантирано.

## 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 **льже**, 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 подготвителните примерни изпити ви затрудняват, значи имате **нужда от още подготовка**. Запишете се на [бесплатния курс по "Основи на програмирането"](#) 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 измежду **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

работка), е необходимо здраво учене. Просто **трябва да намерите време!** Причина #1 за бъксоване 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 **бездаботните** 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 гимназии 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 леката 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 нямаете 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 с **JavaScript** 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 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 бъзовете, 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**, които ще са от полза за развитието ви като програмист:

- <https://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.
- <https://softuni.bg/forum> - официален **форум на СофтУни**. Форумът за дискусии на СофтУни е изключително позитивен и изпълнен с желаещи да помогнат 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.

- <https://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.
- <http://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!
- <http://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.
- <https://fb.com/groups/bg.developers> - групата "Програмиране България @ Facebook" е една от най-големите онлайн общности за програмисти и дискусии по темите на софтуерната разработка на български език във Facebook.
- <https://meetup.com/find/tech> - потърсете **технологични срещи** (tech meetups) около вашия град 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 <http://iteventz.bg> и <http://dev.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 страховити перспективи за развитие и възможности да правите значими проекти с вдъхновение и страсть.

София, 30 май 2018 г.

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: [js-book.softuni.bg](http://js-book.softuni.bg)



## Авторски колектив:

**Бончо Вълков**  
**Венцислав Петров**  
**Димитър Далев**  
**Елена Роглева**  
**Жулиета Атанасова**  
**Захария Пехливанова**  
**Здравко Костадинов**  
**Ивелин Арнаудов**  
**Кристиан Мариянов**  
**Мартин Чаов**  
**Николай Банкин**  
**Николай Костов**  
**Павел Колев**  
**Петър Иванов**  
**Светлин Наков**  
**Стилян Канголов**  
**Християн Христов**  
**Христо Минков**

ISBN 978-619-00-0702-9



9 786190 007029 >